(2)

ETL-0586

New Methods of Change Detection Using Multispectral Data

Charles Sheffield
Gil Richardson

Earth Satellite Corporation
6011 Executive Boulevard, Suite 400
Rockville, MD 20852

May 1991

Prepared for:
U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

E T L

US ARMY ENGINEER TOPOGRAPHIC LABORATORIES
ETL

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE May 15, 1991	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE NEW METHODS OF CHANGE DETECTION USING MULTISPECTRAL DATA		5. FUNDING NUMBERS
6. AUTHOR(S) CHARLES SHEFFIELD AND GIL RICHARDSON		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) EARTH SATELLITE CORPORATION 6011 Executive Boulevard, Suite 400 Rockville, MD 20852		8. PERFORMING ORGANIZATION REPORT NUMBER ETL-0586
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. ARMY ENGINEER TOPOGRAPHIC LABORATORIES ATTN: ETL-SL-T (Don Davis) FORT BELVOIR, VA 22060-5546		10. SPONSORING/MONITORING AGENCY REPORT NUMBER ETL-0586
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report discusses the final phase of a project to develop multispectral change detection methods with emphasis on human activities. Two new algorithms were developed and tested: a Spectral/Spatial Classifier and a Feature Vector Spectral Classifier. The Spectral/Spatial method appears to be a powerful new tool. The Feature Vector results were inconclusive.

14. SUBJECT TERMS			15. NUMBER OF PAGES 128
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

NEW METHODS OF CHANGE DETECTION USING MULTISPECTRAL DATA

FINAL REPORT

PREPARED UNDER
CONTRACT NO. DACA76-86-C-0018

FOR THE

U.S. ARMY ENGINEER TOPOGRAPHIC LABORATORIES
FORT BELVOIR, VA

MAY 1991

TABLE OF CONTENTS

SUMMARY

This report describes the final phase of an on-going project to examine multispectral classification methods, with emphasis on change detection to identify human activities. The first part of this effort, conducted from April to August, 1988, emphasized "conventional" multispectral classification. The emphasis in the present phase of the contract was on new classification and change detection techniques.

Specifically, two new algorithms were implemented and tested on multiple date Thematic Mapper data of the Washington, D.C. area. The methods, described in the body of the report, may be termed Spatial/Spectral (S-squared) analysis, and Feature Vector spectral analysis. The S-squared procedure, at least insofar as the tests performed on this project are concerned, shows spectacular promise. It produced much better results than a conventional multispectral classification. These results are discussed in detail in this report, with associated color imagery.

The Feature Vector spectral analysis gave results, that on the basis of the tests performed on this project, are inconclusive. The technique appears to have potential for detection of man-made features, but was not successful as a change detection tool.

The structure of this report is as follows:

1. Description of the Spectral/Spatial (Complement Approach) method, and the results obtained therefrom.

2. The Feature Vector spectral approach, and its results.

APPENDIX 1:	Source code listing and flow logic for programs developed under this contract.
APPENDIX 2:	Contract deliverables not included in this report; specifically, large photographic products, and magnetic media products.
APPENDIX 3:	Projection operators and band combinations.

ACKNOWLEDGEMENTS

The ideas used in the Complement Approach were derived from project discussions on the problem of small statistics with Don Davis of the U.S. Army Engineer Topographic Laboratories and with Jim Hammack of the Defense Mapping Agency. We would like to thank these individuals and Bob Rand of USAETL for useful discussions during the course of the project, and in particular, thank Don Davis for delineating the logical flow of the Complement Approach algorithm drawn out in Appendix 1.

1. THE COMPLEMENT APPROACH TO SCENE CLASSIFICATION.

This section of the final report describes a new technique for scene classification developed under this contract. It is termed the Complement Approach and it employs both spectral and spatial information contained in image scenes. The technique was developed early in 1990 and employed in practical experiments under this contract, following a series of meetings between contractor and government personnel, in which discussions centered on solving the problem of "small statistics", i.e., on the problem of analyzing within a remotely sensed scene the small features, such as buildings and roads, that are of major practical importance, but which do not represent a statistically large fraction of total scene area. Conventional spectral classification methods normally lose such features by absorbing them into some other class of ground cover.

This section of the report is organized in four parts:

General Approach

Results, including Image Examples

Further Experiments, and

Conclusions.

The source code of all computer programs, developed in the course of the contract and implementing the Complement Approach, is given in **Appendix 1**.

1.1 The Complement Approach to Scene Analysis and Change Detection

1.1.1 Overview.

This technique of scene analysis is designed to find scene elements that occupy few pixels, or partial pixels. It works best when the scene elements of main interest fall into such a category; by contrast, most approaches work well for the majority of pixels in a scene, at the expense of pixels representing "rare" ground cover types (e.g., buildings, roads) as a small percentage of total pixel population.

The general term "Complement Approach" is used since at each stage of analysis, the pixels of main interest are the complementary set to the pixels identified; in other words, the method works by <u>removing</u> pixels from the set to be analyzed. The process is complete when no pixels remain to be analyzed.

Change detection is performed by comparing two scenes for which analysis has been performed. The discussion given here first describes single scene analysis, and then describes two-scene change detection. The approach can utilize varying levels of human expert assistance. We characterize these as:

(1) No expert help;

(2) Expert help from prior map data; and

(3) Expert help by interactive analysis.

These three approaches are all variations of (1), which was employed exclusively on the experiments of this project.

1.1.2 Single Scene Complement Approach

For purposes of analysis, every pixel in a scene to be analyzed is considered to be either a "pure" pixel, representing essentially one type of ground cover with a defined mean and small variance, or a "mixed" pixel, where several different ground covers each occupy part of the pixel. A mixed pixel may be "simple," if the ground covers within it also occur in pure pixels within the scene, of sufficient numbers that their spectral reflectance can be defined by a mean and variance; or a mixed pixel may be "complex," if some ground cover within it does not occur in any pure pixels.

As examples of each category, a pure pixel might be a pixel completely covered with grass or water, in a scene with a good deal of water and grass covers. A simple mixed pixel would be a pixel whose area was part grass and part water. A complex mixed pixel would be part asphalt, part grass, or part asphalt, part grass, and part water, in a scene where no pixels were completely covered by asphalt.

The procedure determines, in order, pure pixels, simple mixed pixels, and complex mixed pixels. It is assumed that no a priori information is available for the spectral signature of material present only in complex mixed pixels. Later, a way is described by which any such a priori information can be incorporated in the analysis.

Stage 1. Select a training area of (presumed) pure pixels. Develop mean and variance/covariance matrix for the number of bands used (e.g., 6 for TM, 4 for MSS) and alarm all pixels in scene at 1-sigma level. Interactively, vary the variance up and down from 1-sigma level to cover a "pure pixel" set. Remove these pixels from the data set considered, and label removed pixels with an appropriate class identifier.

Stage 2. Repeat this procedure for other pure pixel training areas, each time varying (interactively) the variance, then labeling chosen pixels with suitable class identifiers. These pixels are removed from the data set still to be considered.

Stage 3. When all pure pixel data sets have been tagged and removed, the remaining pixels will be simple or complex mixed pixels, or possibly pure pixels of an unidentified class. These remaining pixels should constitute a small percentage of the original pixel set.

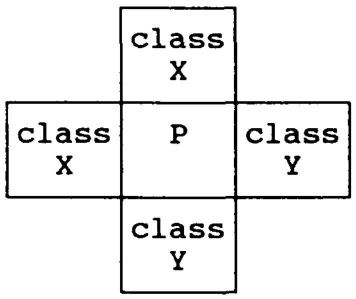
NOTE: To this point, all analysis has been performed using purely spectral properties of the data, and is a conventional classification exercise. It has been described as a supervised operation, but could equally well be unsupervised. Similarly, any type of classification algorithm (e.g., maximum likelihood, parallelepipedon) may be used. However, there is one important difference from standard classification in both execution and intent. Normally, the objective of classification is to assign all scene pixels to pure classes (i.e., each pixel is presumed to be a pure pixel). A "successful" classification is thus one with no unclassified pixels, and to achieve that result the region of spectral space occupied by each pure class is often expanded. When this is done, mixed pixels tend to be assigned to a pure class. Since in the Complement Approach we specifically do not want mixed pixels assigned to pure classes, the variance associated with each pure class will be kept small, and more unclassified pixels than usual will remain when the pure classes have been determined.

To do better than standard classification, it is necessary to invoke more than spectral information about mixed pixels. This additional information is spatial, as described below.

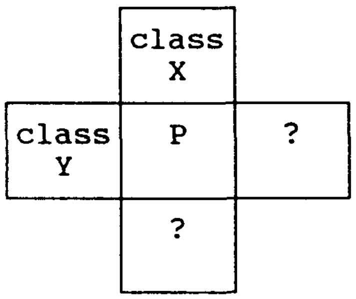
Stage 4. When the pure pixel classes have been identified, classified, and removed from the data set, all remaining pixels are associated with their set of four nearest neighbor pixels. These neighbors may themselves be pure or mixed pixels; in either case, each neighbor is part of an already-classified and removed pixel set, or it is not. The set of pixels not already classified is now divided into two groups: A, the set of pixels at least two of whose neighbors are pure pixels of known classes, and B, the set of pixels no two of whose neighbors are pure pixels of known classes (See **Figures 1 and 2**).

Figure 1: Stage 5 analysis: illustrations.

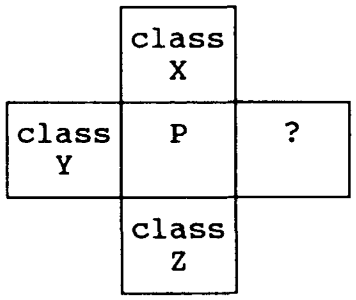
Group A pixels.



Case 1



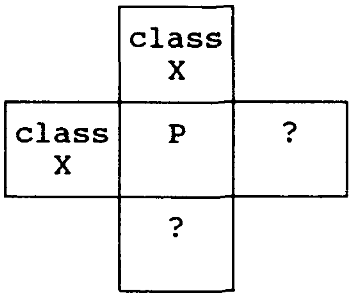
Case 2



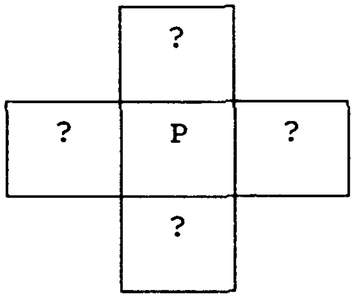
Case 3

Figure 2: Stage 5 analysis: illustrations.

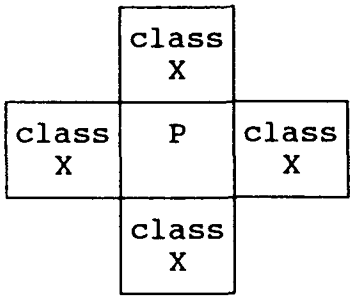
Group B pixels.



Case 1



Case 2



Case 3

Pixels of group A are subject to the next stage of analysis. Analysis of pixels of group B is deferred until later.

Stage 5. Pixels of group A are most likely to be simple mixed pixels, where the mixture consists of only the pure pixel types represented in the nearest neighbors. This assumption is now tested. Let the spectral reflectance mean of class j averaged over the nearest neighbors of any given pixel P be the vector $\vec{C}_j$. In other words, if R of the neighbors of the given pixel are of pure class j,

$$\vec{C}_j = \sum_{r=1}^{R} \vec{c}_r \, /R \qquad (1)$$

or

$$\vec{C}_p = \sum_{j=1}^{J} \alpha \vec{C}_j$$

where $\vec{c}_r$ is the vector of reflectances for each spectral band (so this would be a vector with 4 components for MSS data, 6 for TM data excluding thermal I/R).

Now if P is a mixture of J neighbor classes, we will have

$$\vec{C}_p = \alpha_1 \underline{\vec{C}_1} + \alpha_2 \underline{\vec{C}_2} + \ldots + \alpha_J \underline{\vec{C}_J} \tag{2}$$

If we consider only the 4 nearest neighbors of P, we have $J \leq 4$ (i.e., only 4 pure classes, at most, can be present in the nearest neighbors). Further, each coefficient α_j must be non-negative, since a positive area must be covered by any class present in a pixel.

We treat (1) as a set of linear equations which determine α_j by solution of equations if $J=4$ and $\vec{c}_p$ has four components, by least squares otherwise (always possible for TM data since there are at least 6 equations and at most 4 pure neighbor classes). **See Notes A and B.**

Now either the fit (2) is achieved to some satisfactory tolerance for pixel P, or it is not. If (2) is satisfied, pixel P is a simple mixed pixel, and it is labeled with the classes it contains and removed from the set A. If no satisfactory fit is achieved by (1), P remains for further analysis. This may be because neighbors of P are simple mixed pixels, and it was therefore not considered in the first pass through group A pixels. The process must thus be iterated, to see if any pixels now have two or more known neighbors of simple mixed type, in addition to neighbors of pure pixel type.

Stage 6. The remaining (i.e., not meeting tolerances) pixels in group A and the pixels in group B are aggregated to form a single set. This set is now divided into two groups: A^1, the set of pixels at least two of whose neighbors are pure pixels of known classes or simple mixed pixels as determined in Stage 5; and B^1, the set of pixels no two of whose neighbors are pure pixels of known classes or are simple mixed pixels as determined in Stage 5.

Stage 7. As before, the assumption to be tested is that pixels in group A^1 are simple mixed pixels, where now the mixture consists of pure pixel types represented in nearest neighbors, and of classes represented in known simple mixed pixels that are nearest neighbors (**See Figures 3 and 4**). However, the analysis method used in Stage 5 must be modified, because now the neighbors of any pixel, P, in group A^1 may be simple mixed pixels rather than pure pixels. (Note that if the nearest neighbors of P are pure pixel types only, the analysis for them will be identical to that of Stage 5. It need not be repeated, and such pixels may be placed at once into group B^1, since they must have failed the test during Stage 5 analysis.)

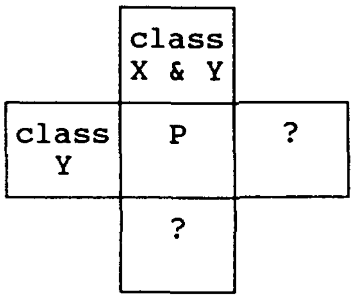
At this point an assumption is introduced to assure determinancy of subsequent calculations. No more than four classes will be permitted as components in the neighbors of P. If more than four classes are present, because the neighbors are mixed pixels, P will be placed in class B^1.

The analysis to determine if pixels of A^1 are simple mixed pixels now resembles that of Stage 5, but differs in one important respect. Instead of using computed local reflectance means of the nearest neighbors of P, the class means must be used for classes present in the neighbors of P, as either pure or simple mixed pixels. This is less good than using local pure pixel reflectance means, but it is necessary because now the pixels in the group A^1 must have as neighbors mixed pixels as well as possible pure pixels (the case of all pure pixel neighbors was already disposed of in Stage 5).

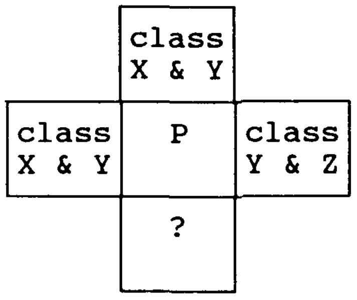
If the neighbors of P contain J classes ($J \leq 4$, by the restrictive assumption already made), we seek a fit of the form:

Figure 3: Stage 7 analysis: illustrations.

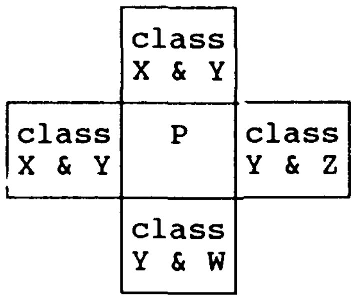
Group A' pixels.



Case 1



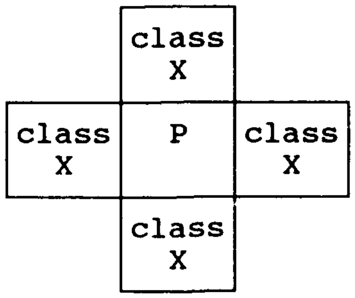
Case 2



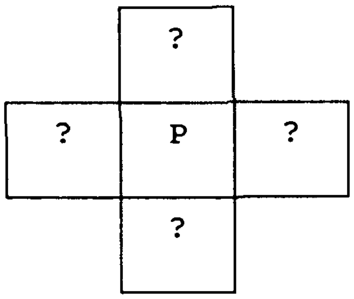
Case 3

Figure 4: Stage 7 analysis: illustrations.

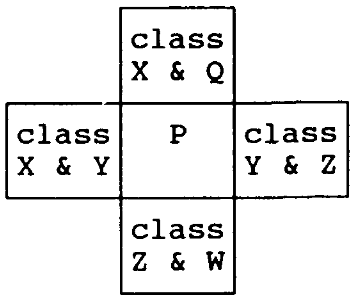
Group B' pixels.



Case 1



Case 2



Case 3

$$\vec{c}_P = \alpha_1 \cdot \underline{\vec{c}_1} + \alpha_2 \cdot \underline{\vec{c}_2} + \ldots \alpha_j \cdot \underline{\vec{c}_j} \tag{3}$$

where now $\underline{\vec{c}_1}$; $\underline{\vec{c}_2}$ are <u>class</u> mean spectral reflectances.

The solution by least squares proceeds as before. Either the fit of (3) is achieved to some tolerance, or it is not. If it is, then pixel P is a simple mixed pixel, and it is labeled with the classes it contains and removed from the set A^1. If not, pixel P is added to the set B^1.

<u>Stage 8</u>. Stage 7 may now be repeated, splitting the set B^1 into two parts, A^2 and B^2. In this case, A^2 is the set of pixels whose neighbors are pure pixels or simple mixed pixels (repeating Stage 7 may yield new mixed pixels, since the first time through, simple mixed pixels identified in Stage 7 were previously of unknown type).

The iteration should be continued, until no new pixels are classified. The process will always converge, since each iteration removes pixels from the A set, and never restores them.

<u>Stage 9</u>. At this point, all identifiable pure pixels and simple mixed pixels have been tagged and removed from the total pixel set. The remaining pixels may be pure or simple mixed pixels of unidentified classes, or complex mixed pixels containing materials present only as partial mixed pixels. In a traditional classification exercise, these pixels, hopefully few in number, are regarded as undesirable noise, or as a failure of the classification process. However, in seeking point or line information for cartographic use, this residual data set is likely to contain the scene elements of most interest. It is the broad, statistically dominant regions of

pure pixels which are of less interest, and have been removed from consideration in the previous analysis.

The unidentified pixels in the $B^{(N)}$ set are now divided into two groups: S, when one or more nearest neighbor pixels belong to the complementary set of pixels already classified; and T, when all neighbor pixels are themselves unidentified. If most pixels of the $B^{(N)}$ set are of group T, it is likely that the variances permitted in the first classification of Stages 1 and 2 were too small. The analysis can be repeated with more generous variances. However, there is then also a danger of throwing away (by classifying them as pure pixels) complex mixed pixels of real interest.

A better method is the examination of pixels in the T-set, to see if they constitute a new and unidentified cluster (i.e., a new pure pixel set) with tight statistical grouping. If so, this class is added to the pure classes and Stages 3-9 are repeated. (This alternative was not explored in the current effort.)

Stage 10. Analysis of group S pixels is now performed. At this point, a priori information on the spectral reflectances of small targets that may be within the scene can be utilized.

The assumption is that the pixels of group S are complex mixed pixels, with one or more (but only one per pixel) unidentified materials present at the sub-pixel level. For example, a concrete road may be present in a scene, of a width small enough that no pixel is all concrete. To test the assumption, we ask if the spectral reflectances of a pixel P in group S can be made up of a positive-weight combination of the reflectances present in the pixels that neighbor it, plus some amount of an added class (i.e., material) of unknown nature, or of some class whose reflectances are provided a priori.

Thus, we seek a relationship of the form

$$\vec{c_p} = \alpha_1 \vec{c_1} + \alpha_2 \vec{c_2} + \ldots + \beta \vec{X} \qquad (4)$$

or

$$\vec{C_p} = \sum_{j=1}^{J} \alpha_j \vec{C_j} + \beta \vec{X}$$

where α_1, α_2...and β are unknown weight coefficients, $\vec{c_1}$, $\vec{c_2}$ ...are known vectors of spectral reflectances, for the classes occurring in the neighbors of P; and $\vec{X}$ is the vector of spectral reflectances for an unknown, or a priori assumed, extra material present at the sub-pixel level.

Now let us count unknowns and equations. In order to be specific, assume we are working with 6-band TM data, and as before, limit the number of classes permitted in the neighbors of P to 4 or less. Assume that the vector $\vec{X}$ is unknown. Then a single pixel P yields 6 equations (one for each band reflectance) and up to 11 unknowns (coefficients α_1...α_4, β, and six unknown components of $\vec{X}$). However, the components of $\vec{X}$ are the same, or close to the same, in each pixel. Thus if we have M pixels with some of the material of spectral reflectance $\vec{X}$ within each of them, there will be 6M equations and at most 5M+6 unknowns. This is thus a determinate situation for $M \geq 6$.

In the case where there is a priori information, so that the spectral reflectance $\vec{X}$ of a material present at sub-pixel level is known, then each pixel P contains no more than 5 unknowns ($\alpha_1 \ldots \alpha_4$, β), and one pixel provides 6 equations. Thus the proportion of $\vec{X}$ present in P can be estimated from a single pixel. (See also Note B.)

As before, the solution to (4) by least squares is acceptable only if $\alpha_1 \ldots \alpha_j$, β, $\vec{X}$ all have positive values.

If several unknown materials are present at sub-pixel level, the problem is more complicated. It is necessary to specify image regions in which only one unknown material is likely to occur. This information may be provided by analyst action, or by a priori (cartographic) information about the contents of a scene.

When Stage 10 has been performed for each pixel ($\vec{X}$ assumed known) or groups of 6 or more pixels ($\vec{X}$ assumed unknown) in group S, pixels for which the fit (4) is within a tolerance are now tagged as containing fractions of a new material of spectral reflectance vector $\vec{X}$.

All such pixels are now displayed, and examined to see if they exhibit point or line characteristics.

1.1.2.1. COMMENTS

1. The approach proposed here forms a basic methodology to tackle the "small statistics" problem encountered whenever scene targets are present as only a few pixels, or only as partial pixels. Since the approach is new, the stages described are likely to need revision based on experience, or at least careful evaluation, before a standard program module can be routinely used.

2. Of the required tools, Stages 1 and 2 were standard and already existed. Stages 3 through 10 required for program development under this contract.

3. The reason that Stages 3 through 10 are not in standard classification packages is their simultaneous use of spectral and spatial scene information, and their emphasis on the small statistics aspects of scene classification.

1.1.2.2. NOTES

NOTE A: The least squares solution for α_j

Write α_1, $\alpha_2 \ldots \alpha_j$ as the vector $\underline{\alpha}$, of J components ($J \leq 4$).

Then,

$$\vec{c}_p = \sum_{j=1}^{J} \alpha_j \vec{c}_j$$

or in matrix terms,

$$C_i = \sum_j C_{ij} \alpha_j$$

C is a matrix of B rows and J columns, where B = number of spectral bands and J = number of classes in nearest neighbors.

Since $J \leq B$, $(C^T C)$ is non-singular, of dimension JxJ.

The least squares solution of the set of equations is given by:

$$\vec{\alpha} = (C^T C)^{-1} C^T \vec{C}_p$$

where $C = (c_{ij})$.

NOTE B: <u>Interpretation of the coefficients α_j, β</u>

For the pixel P, we made a fit of the form:

$$\vec{c}_p = \sum_j \alpha_j \vec{c}_j + \beta \vec{X}$$

What is the meaning of the coefficients in this fit? If a pixel has area A and spectral reflectance $\vec{c}_j$, then the spectral reflectance per unit area is $\vec{c}_j / A$. An area α_jA will contribute spectral reflectance $\alpha_j \vec{c}_j$. Thus if a pixel is made up of different materials, the total spectral reflectance will be

$$\sum_j \alpha_j \vec{c}_j + \beta \vec{X}$$

$$\textit{where} \quad \sum_j \alpha_j + \beta = 1$$

since the sum of areas α_j A and βA must be A. The condition that the α's and β sum to unity can be imposed in the fitting procedure. However, it is desirable to experiment with and without this condition.

It is particularly dangerous to seek to impose the unit sum condition when $\vec{X}$ comes from an a priori reflectance. The actual observed reflectances include sun angle and atmospheric effects, and these serve as a (usually known) multiplier on $\vec{X}$. Thus the condition that the coefficients sum to unity will no longer be fulfilled.

1.1.3 Change Detection Using the Complement Approach

General Comment

One popular approach to change detection proceeds first by creating a difference image, band by band, then analyzes that image, usually by some form of multispectral classification. Such a method has the advantage of economy, since the classification, which is usually expensive of processing time, is done only once.

The use of differences images is not appropriate for change detection using the Complement Approach, since the signatures of different ground types within the scene are needed to find partial pixels. To see this by example, suppose that there are available two scenes at different dates of the same ground area. Suppose that the whole scene consists of three types of ground cover: grass, water, and concrete. If the images of the two different dates are differenced, band by band, the resulting difference scene will contain the following possible types of pixels:

(1) pure unchanged (grass remains grass, water remains water, concrete remains concrete)

(2) pure changed (grass to water, water to concrete, etc.)

(3) mixed pixels.

From the difference image alone, there is no way of knowing if the mixed pixel was mixed on both scenes, or if it is pure on one, and mixed on the other. Further, since there will be small sub-pixel off-sets of one date from the other, the fraction of pure pixel types within each mixed pixel will be different on the two dates. Finally, there is no way of relating pixels to their neighbors in any meaningful way.

The method of scene differencing before classification is therefore not appropriate when using the Complement Approach. The basic reason is that a statement of "No change" from one date to another, as provided in analysis of a difference image, lumps together to a single pixel class all the information, "Grass remains grass, water remains water, concrete remains concrete, etc.," obtained when the two dates are analyzed separately.

<u>Procedure for Change Detection with the Complement Approach</u>

Each date is processed using the method described in Section 1.1.2. The result of such processing is a scene in which pixels fall into one of the following categories:

(1) pure pixels, of a class which is described by a class identifier (e.g., a letter or number). Note that the class has not been described in <u>physical</u> terms, such as "grass" or "water", unless operator involvement has provided such labels;

(2) simple mixed pixels, made up of partial pixels of known pure classes, each class described by the same class identifiers as in the pure pixels;

(3) complex mixed pixels, made up of partial pixels of known pure classes, and new classes of materials present only at the sub-pixel level;

(4) pixels that cannot be classified. Such pixels may be pure pixels of unidentified classes, or pixels that because of other factors cannot be classified (e.g., anomalous terrain aspect angle, local haze or cloud, or any other reason that causes a pixel to fail the chosen tolerance tests).

Both dates possess such classifications of their pixels. The task now falls into two parts:

(1) <u>Pure class matching</u>. The pure classes of Date 1, which we will label A, B, C, etc., must be matched with the pure classes of Date 2, which we label as A', B', C', etc. This is straightforward, because most pure pixels of Date 1 remain as pure pixels in Date 2.

Following class matching, pure pixels that have not changed are tagged as "No change" pixels. Pure pixels that have changed are flagged as "Change" pixels, with the appropriate pair of class identifiers (e.g., B to C').

(2) Partial pixel matching. A pixel that is a partial pixel (simple or complex) on Date 1 will usually remain a partial pixel on Date 2, although the fraction of the class types within it will normally change because of sub-pixel shifts in pixel location.

Simple and complex partial pixels will be analyzed thus:

(1) Simple partial pixels that contain the same pixel class types on Dates 1 and 2 will be tagged as "No change" pixels.

(2) A pixel that changes from pure to simple, or simple to pure, will be tagged as "No change" provided that the pure pixel class type is present on both dates.

(3) A pixel that changes from pure to simple, or simple to pure, will be tagged as "Change" when any pixel class type is different on the two dates.

(4) A partial pixel on both dates that contains different pixel class types on Dates 1 and 2 will be tagged as a "Change" pixel, regardless of whether those classes are of pure type or not.

However, the way in which the tagging is done must be different when the pixel on either Date 1 or Date 2 is a complex mixed pixel. This is necessary since a complex mixed pixel contains material that is present only at sub-pixel level; thus with a complex mixed pixel, a sub-pixel shift in pixel location can be enough to convert it to a simple mixed pixel or a pure pixel (e.g., a shift of a few meters in pixel location can take us off a road completely).

The tagging of partial pixels with different classes on Dates 1 and 2 should therefore take one of the two forms:

(1) Simple mixed pixel, change;

(2) Complex mixed pixel, possible change.

By alarming only the second category on a screen, these pixels are examined to see if they are border roads, large buildings, etc. If so, they are probably in reality "No Change" pixels.

NOTE: Change detection employing this approach was not implemented under the contract.

1.2 Results Using the Complement Approach

1.2.1. Geometrical viewpoint.

The mathematical techniques defined in Section 1.1 above become clearer when they are described in geometrical terms. Before presenting the results of numerical experiments in scene classification, we provide the geometric interpretation that makes those experiments easier to visualize. Although both supervised and unsupervised classification experiments were performed on this contract, the basic geometric viewpoint applies equally to both, and the discussion is therefore provided for supervised classification only.

Consider the representation of the pixels of a scene in spectral space, with the grey levels in each band assigned to one spectral axis. Drawing only two bands for the ease of presentation, an example of such a pixel set is shown in **Figure 5.a**. Every pixel occupies a unique location in spectral space, with its coordinates along each band axis being equal to the grey level in that band. Although more than one pixel may be at the same location, each pixel is at one unique location.

Usually, the display in spectral space shows clustering into a number of groups or "blobs" with a loosely-defined center. For example, in Figure 5.a there are clearly three main groups of pixels. The whole purpose of classification methods is to recognize such pixel groupings, and then to place as many pixels of the scene as possible into the groups, usually termed "classes". The success or failure of a classification algorithm is measured by how well it places the pixels of the scene into such well-defined and separable classes.

Figure 5.b shows a spectral clustering of pixels into three classes. In this example, the clusters are themselves well-defined, but a significant number of pixels remain outside the clusters, i.e., they are unclassified. This is a very typical result of applying a classification program. To classify these remaining, unclassified pixels, conventional classification methods do one of two things: either they increase the number of groups (classes), or they increase the size of each group by being more generous in defining permitted group variances (i.e., the overall dimensions of the group in spectral space). These two approaches are illustrated in **Figures 5.c and 5.d**.

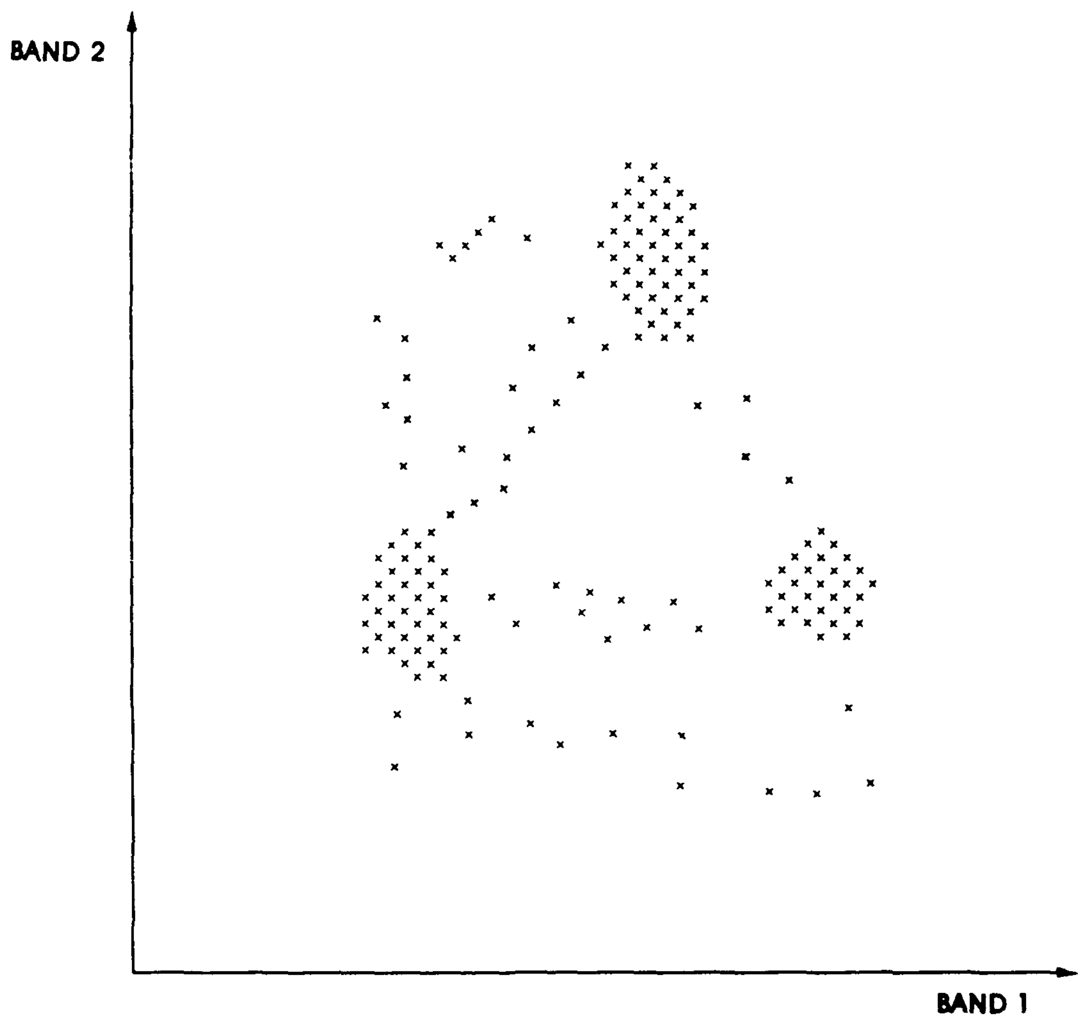


ILLUSTRATION SHOWS ONLY 2 BANDS FOR SIMPLICITY.

FIGURE 5.a: Conventional multispectral classification

Conventional multispectral classification, either supervised or unsupervised, seeks to place all scene pixels (Figure 5.a) into a small number of groups (usually termed "classes") with defined means and small variances (Figure 5.b). Pixels which do not readily fall into such classes, perhaps because they are a mixture of classes, are accommodated in one of two ways: (a) by creating new classes (Figure 5.c) or (b) by increasing the tolerance in existing classes (Figure 5.d).

In the case of mixed pixels, both these procedure are inappropriate. They lead either to a successive number of classes, or to diffuse, large classes of high variance, accepting as members pixels which do not belong to them.

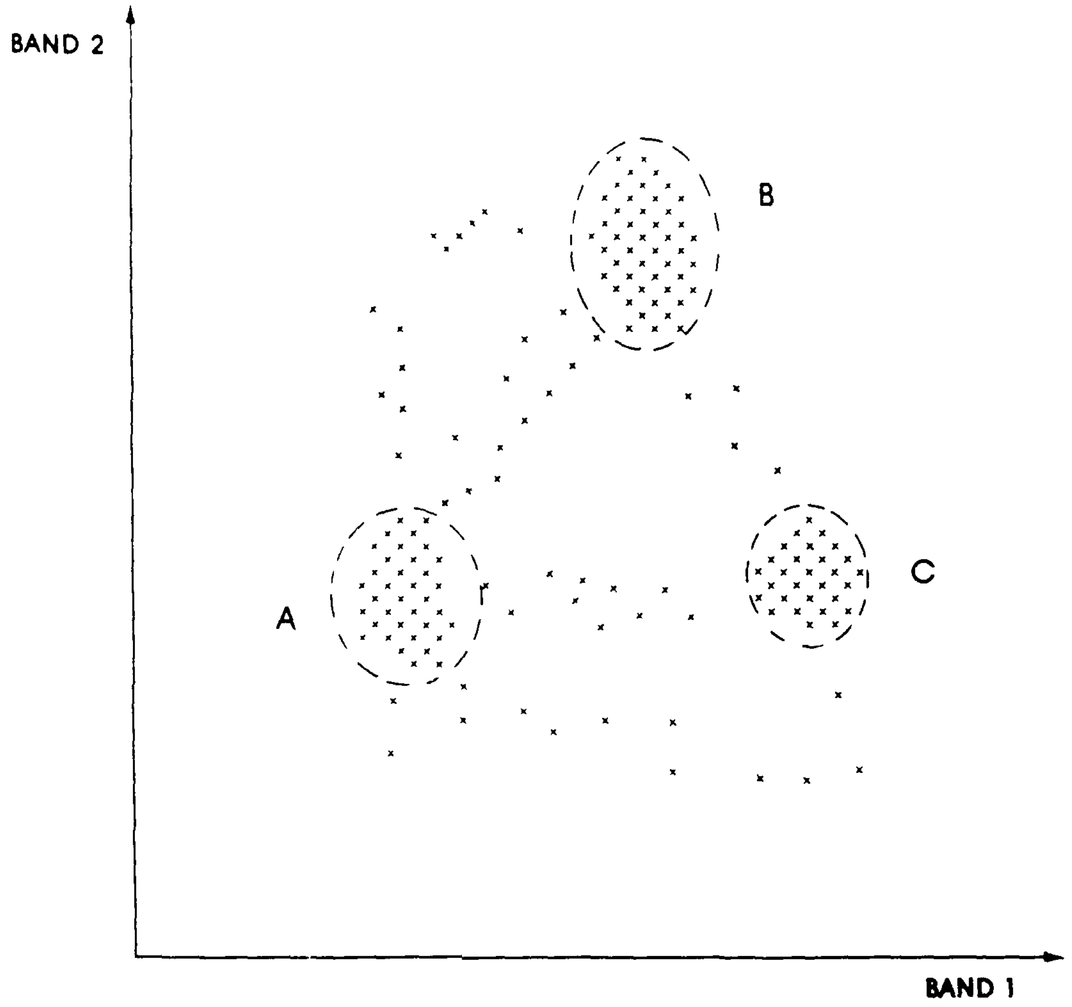


ILLUSTRATION SHOWS ONLY 2 BANDS FOR SIMPLICITY.

FIGURE 5.b

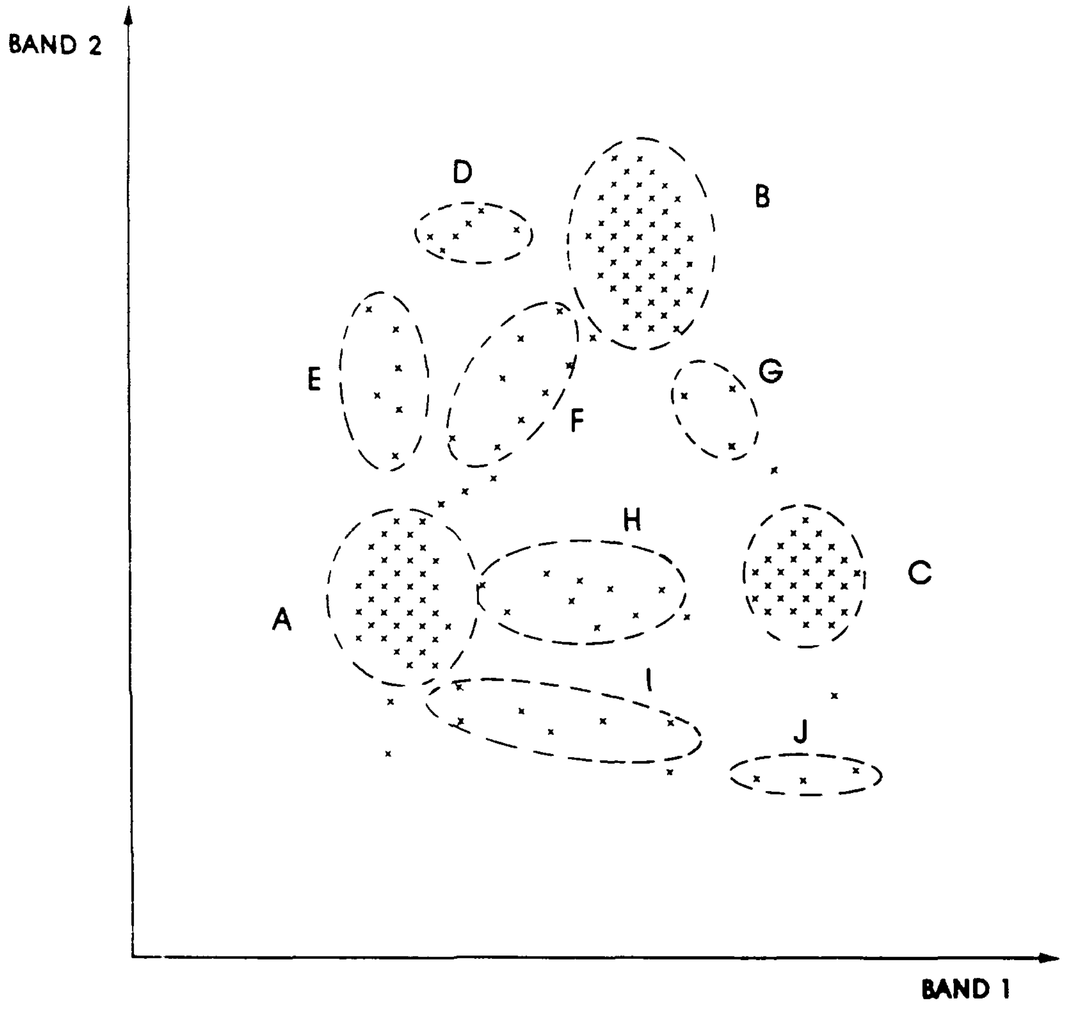


ILLUSTRATION SHOWS ONLY 2 BANDS FOR SIMPLICITY.

FIGURE 5.c

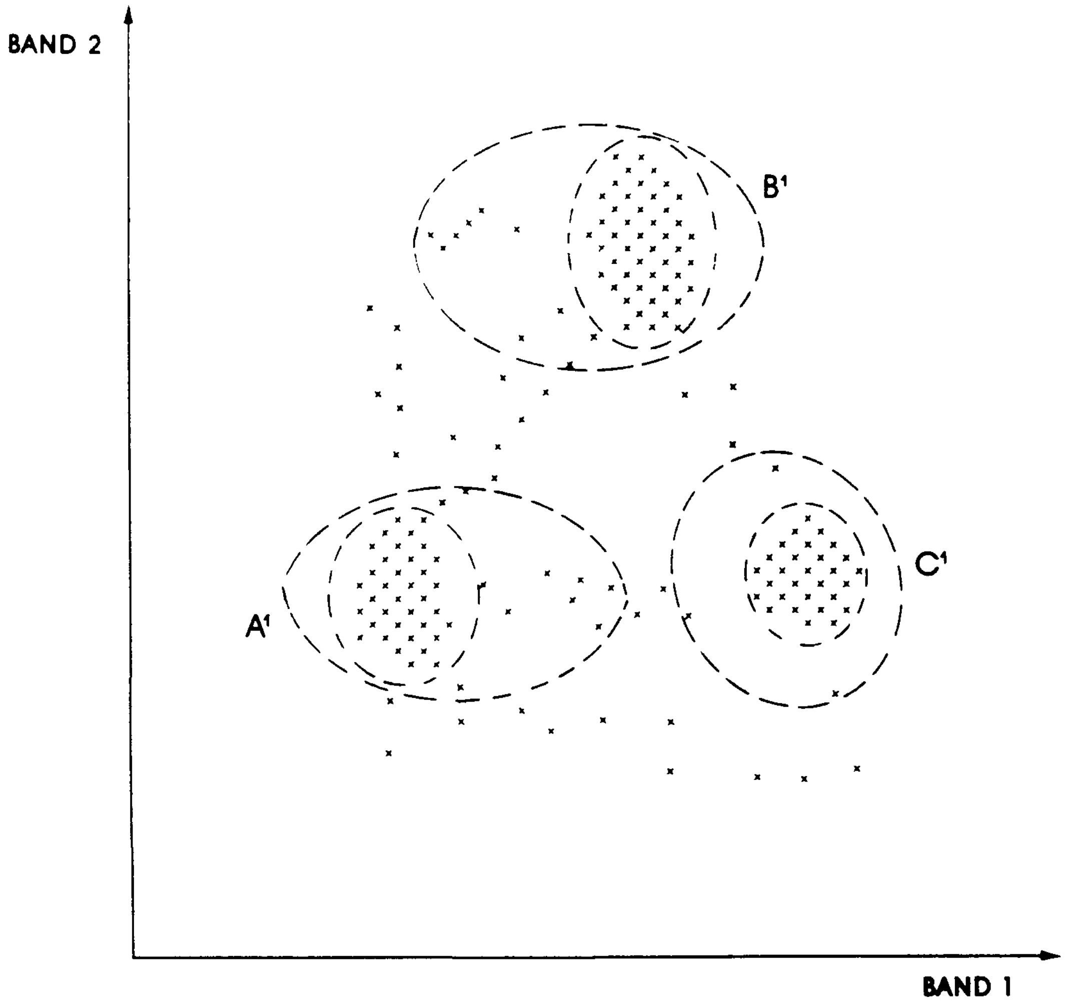


ILLUSTRATION SHOWS ONLY 2 BANDS FOR SIMPLICITY.

FIGURE 5.d

However, in tackling the problem of small statistics, both these methods have a major flaw: they tend to absorb into existing or new classes exactly those pixels, statistically small in number compared with the rest of the scene, which may contain interesting elements such as roads or buildings. These are pixels which it is specifically desired to keep separated from the rest of the image, for detailed study.

In addition, Figure 5.b makes it clear that a significant number of the unclassified pixels in the example lie close to lines that connect the centers of major pixel groups. Such pixels are almost certainly mixed pixels, containing ground cover elements of two known groups. **Figure 6** shows that an algorithm able to deal with such mixed pixels should do a significantly better job of classification than the "pure pixel" spectral classifier normally used.

1.2.2 The use of spatial information.

Given a pixel that has not been classified into one of the "pure" pixel classes already identified, one could argue that the next thing to do is to test to see if such a pixel is made up of a mixture of classes of known pure type, i.e., a "simple mixed pixel." This will be the case if that pixel's reflectance in each spectral band can be written as a positive linear combination of the spectral reflectances of any of the already-known classes. Such an approach suffers from a new problem: with many classes, the number of possible class combinations grows extremely rapidly. For N classes, there are $N(N-1)/2$ class pairs to be considered, and $N(N-1)(N-2)/6$ class triplets. Thus if $N = 10$ which is not a large number of classes for normal classification work, we have 45 class pairs and 120 class triplets to consider. And although the probability of being able to make any given set of pixel reflectances as a combination of known classes grows fast, the chance that such a combination is meaningful rapidly decreases.

What is needed is additional information that can limit the classes for consideration in constructing the simple mixed pixels. Such information is provided by spatial considerations. Thus far, the classification has been made purely on the basis of spectral properties of the pixels. However, each pixel has its own set of nearest neighbors, and the chance that a pixel

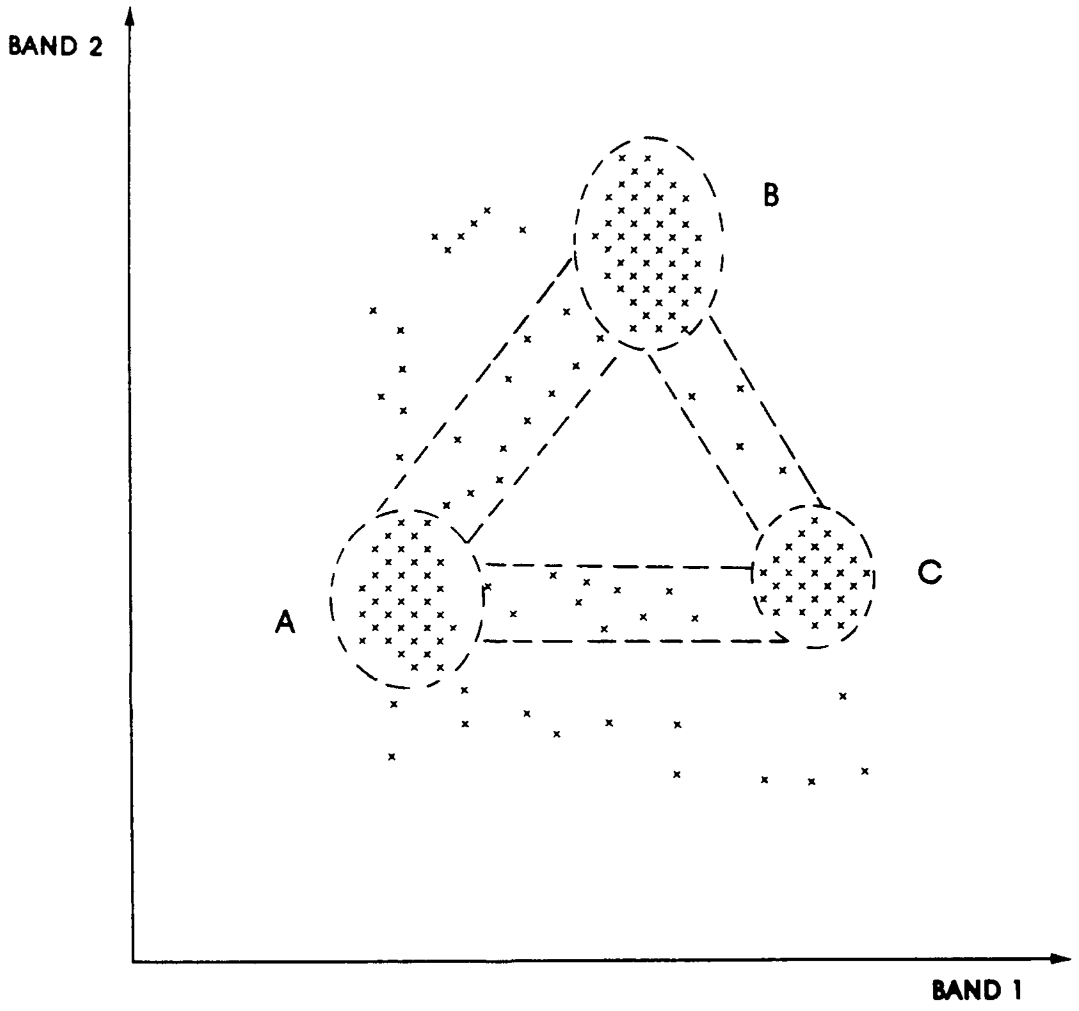


ILLUSTRATION SHOWS ONLY 2 BANDS FOR SIMPLICITY.

FIGURE 6: Spatial spectral classification and the use of mixed pixels

Beginning with a small number of classes of known means and small variances, an initial classification is performed. Following this, every unclassified pixel in the scene is examined to see if it is a combination of classes of its neighbors. Geometrically, pixels are examined to see if they lie on or close to lines or planes joining the means of existing classes, and lying within the subspace spanned by those class means.

The original classes remain tightly defined and have low variance. Pixels which are not classified as pure or mixed pixels become candidates as pure or mixed pixels involving an as-yet undefined class.

is a combination of classes that do not occur in the pixels that neighbor it is very small. This observation formed the basis for the approach detailed in the previous section, and defined as the Complement Approach. Spectral/spatial, or S-squared classifiers, are compared in the following sections with conventional spectral classifications, which are termed S classifiers.

1.2.3 Experiments with S and S-squared classifiers.

In order to permit a large number of experiments to be performed at tolerable cost in machine time, most runs were made on a particular sub-scene window (186 x 157 pixels beginning at Row 325, Column 603) of a Thematic Mapper scene around Washington D.C. This window forms one of the regions of the 2,048 x 2,048 mosaic shown in **Plates 1 through 5**, which are all TM data of different dates. Only the most successful or interesting of the procedures employed on the window were applied to the larger mosaic and only for May 1985, and May 1987.

1.2.3.1 Initial experiments.

The first experiments were performed using both supervised (maximum likelihood) and unsupervised (Euclidean distance) classification methods. Three facts became apparent during those first sets of computer runs.

First, the numerous different possible combinations of program parameters such as sigma levels, fitting tolerances, and number of classes with either supervised or unsupervised classifiers would force early decisions on some of those parameters. Thus, although initial runs were performed using as few as two (2) and as many as twenty-five (25) classes, the final and more systematic experiments were all done using from 2 to 7 classes, and only three levels of fitting tolerance were explored.

Second, fully unsupervised classification was not going to be a satisfactory approach, given the limited time and budget available. Very few pixels were classified using the unsupervised classification program, unless either a large number of classes or very generous class variances were used. In addition, interpretation of the classes defined in the unsupervised mode was often difficult. A decision was finally made to restrict experiments to supervised maximum likelihood classifications.

Third, initial experiments with the mixed pixel spectral/spatial (S-squared) technique revealed one unanticipated theoretical problem with the original algorithm, as follows: The algorithm assumed in looking for mixed pixels that two or more different classes of the mixed pixel would be present in the pixel's neighbors. In practice, pixels existed that were entirely surrounded by a single class of pixel (notably water) and hence the classification of that pixel did not take place.

To allow for this possibility, a new program was developed which examines all unclassified pixels with two or more neighbors of a single known class, and asks if they are a mixture of that class and some other known class. If so, these pixels are added to the category of classified mixed pixels. Note that for N classes, this algorithm must examine (N-1) possible mixtures of pure classes.

1.2.3.2 Main experiments.

Once the spectral classification technique and appropriate number of classes had been decided upon, and the necessary added software described above developed, the procedure employed in all later runs of the programs was as follows:

1) Define with operator interaction an initial set of (hopefully pure) classes within the window area, and develop the spectral statistics for each class.

2) Perform a purely spectral scene classification (S classifier), setting some prescribed tolerance (sigma-level) for acceptance of pixels into the defined classes. The statistical significance levels employed were 1-sigma and 3-sigma.[1]

3) Display the S-classification results .

[1]For data that follow a normal distribution, these sigma levels correspond to pixel acceptance at the 68% probability level and the 99.74% level, respectively. It should be noted that use of a value as high as 3-sigma almost guarantees that pixels will be accepted into a group that do not really belong there.

4) Run the mixed pixel spectral/spatial (S-squared) classifier, iterating it until no further pixels are classified.

5) Display the S-squared classification results, comparing them with the Step 3) results.

6) Repeat steps 1) - 5), varying the scene sigma-levels, the number of initial classes, the tolerance of the fit, or all three.

1.2.4 Results

1.2.4.1 S-classification.

As a "base case" for comparison purposes, a standard maximum likelihood supervised classification was run at the 1-sigma and 3-sigma levels. This was done for two different dates (May 1985 and May 1987). The 1-sigma case was run using from 2 to 7 classes, since the mixed pixel runs would also be performed with 2 through 7 classes and sigma = 1. The 3-sigma case was run for 7 classes only, in order to compare the 1-sigma S-squared classification with the 3-sigma S-classification.

The results of these runs are shown in **Tables 1 and 2**, for 1985 and 1987 respectively. The following observations can be made after inspection of those results:

(1) The conventional spectral classification with sigma = 1 leaves most of the scene unclassified (82% for 1985 case, 77% for 1987 case) even when 7 classes are used. The classification success is lower with less classes.

(2) Even with sigma = 3 (where the probability that a pixel <u>not</u> be assigned to a class is only .26%), the conventional spectral classification with 7 classes leaves 18% of the scene (1985) or 9% of the scene (1987) unclassified.

TABLE 1

MAY 1985

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	24683	84.53
1	Tree	2677	9.17
2	Water	1842	6.31

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	24505	83.92
1	Tree	2677	9.17
2	Water	1842	6.31
3	Runway	178	0.61

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	24311	83.25
1	Tree	2677	9.17
2	Water	1842	6.31
3	Runway	178	0.61
4	Urban	194	0.66

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Classified	24204	82.88
1	Tree	2677	9.17
2	Water	1842	6.31
3	Runway	178	0.61
4	Urban	194	0.66
5	Veg	107	0.37

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	24144	82.68
1	Tree	2677	9.17
2	Water	1842	6.31
3	Runway	178	0.61
4	Urban	194	0.66
5	Veg	107	0.37
6	Airfield	60	0.21

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	24084	82.47
1	Tree	2677	9.17
2	Water	1842	6.31
3	Runway	178	0.61
4	Urban	194	0.66
5	Veg	107	0.37
6	Airfield	60	0.21
7	Newbare	60	0.21

May 1985, Standard Deviation = 3.0

Number	Name	Count	Percent
0	Unclassified	5325	18.24
1	Tree	10559	36.16
2	Water	6239	21.36
3	Runway	1220	4.18
4	Urban	1594	5.46
5	Veg	3466	11.87
6	Airfield	263	0.90
7	Newbare	536	1.84

TABLE 2

May 1987

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	22958	78.62
1	Tree	3755	12.86
2	Water	2489	8.52

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	22856	78.27
1	Tree	3755	12.86
2	Water	2489	8.52
3	Runway	102	0.35

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	22658	77.59
1	Tree	3755	12.86
2	Water	2489	8.52
3	Runway	102	0.35
4	Urban	198	0.68

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	22503	77.06
1	Tree	3755	12.86
2	Water	2489	8.52
3	Runway	102	0.35
4	Urban	198	0.68
5	Veg	155	0.53

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	22463	76.92
1	Tree	3755	12.86
2	Water	2489	8.52
3	Runway	102	0.35
4	Urban	198	0.68
5	Veg	155	0.53
6	Airfield	40	0.14

Maximum Likelihood, Standard Deviation = 1.0

Number	Name	Count	Percent
0	Unclassified	22366	76.59
1	Tree	3755	12.86
2	Water	2489	8.52
3	Runway	102	0.35
4	Urban	198	0.68
5	Veg	155	0.53
6	Airfield	40	0.14
7	Newbare	97	0.33

May 1987, Standard Deviation = 3.0

Number	Name	Count	Percent
0	Unclassified	2765	9.47
1	Tree	10385	35.56
2	Water	7168	24.55
3	Runway	630	2.16
4	Urban	3258	11.16
5	Veg	3849	13.18
6	Airfield	214	0.73
7	Newbare	933	3.19

1.2.4.2 S-squared classification.

Tables 3 through 8 show the results of the S-squared classification, all with sigma = 1, for 2 through 7 classes. These runs were performed for three different values of the least squares tolerance parmeter, T = 5, 10, and 15. The value of T can be thought of geometrically as the distance that a pixel can be "off the line" connecting pure spectral class means, and still be accepted as a simple mixed pixel.

The results show a striking improvement in classification over the spectral S-classification. Even with T = 5, which is a "tight" value of the tolerance, the unclassified portion of the image is only 19% (1985) or 21% (1987). With a more realistic value of T = 10, this goes down to 7% (1985) and 8% (1987).

The images resulting from these classifications are shown in **Plates 6 through 29**[2]. In each case, the unclassified parts of the image are shown in black. These are therefore, according to the Complement Approach, the "interesting" parts of the image that remain for further analysis. However, it is also important that the classification performed by the new S-squared classifier make sense, as thematic classes. Inspection of the images shows that this is the case.

It is instructive to compare the 7-class S-classification with sigma = 1 and sigma = 3 with the 7-class S-squared classification with sigma = 1 and T = 5, 10, and 15. These are shown on **Plates 9, 13, 17, 21, 25 and 29.** In every case, the runs with sigma = 1 for the S-squared classifier appear better than the corresponding runs with sigma = 3 for the conventional S-classifier.

This is a surprising result. It indicates that, for this image at least, <u>most pixels are mixed pixels</u>. Consider, for example, **Plate 13**. Conventional maximum

[2]These plates and **Plates 30 through 35** are provided in the body of the report reduced to a size convenient for report binding, and are not at any precise scale. However, the same information has been separately provided as contract deliverables in larger format. It is recommended that these products be used whenever <u>details</u> of a particular classification result are of concern, rather than a general comparison of different methods.

TABLE 3

May 1985

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	13604
Tree	7648
Water	7950

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	10414
Tree	10150
Water	8386
Runway	252

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	9496
Tree	10434
Water	8419
Runway	452
Urban	401

Mixed Pixel, Least Squares Errorr = 5

CLASS	COUNT
Unclassified	6608
Tree	11505
Water	8437
Runway	566
Urban	479
Vegetation	1198

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	5826
Tree	11569
Water	8425
Runway	570
Urban	537
Vegetation	1118
Airfield	536

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	5466
Tree	11574
Water	8456
Runway	600
Urban	523
Vegetation	1082
Airfield	496
Newbare	209

TABLE 4

May 1985

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	10076
Tree	10738
Water	8388

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	7233
Tree	12584
Water	8713
Runway	672

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	5557
Tree	13175
Water	8723
Runway	906
Urban	841

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	3247
Tree	13228
Water	8784
Runway	866
Urban	899
Vegetation	1702

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	2269
Tree	13261
Water	8816
Runway	695
Urban	905
Vegetation	1506
Airfield	954

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	2269
Tree	13261
Water	8816
Runway	695
Urban	905
Vegetation	1506
Airfield	954

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	2149
Tree	13238
Water	8799
Runway	716
Urban	883
Vegetation	1414
Airfield	736
Newbare	338

TABLE 5

May 1985

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	8387
Tree	11818
Water	8647

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	5470
Tree	13592
Water	8748
Runway	1392

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	4056
Tree	14118
Water	8759
Runway	935
Urban	1334

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	2301
Tree	13608
Water	8839
Runway	1058
Urban	1071
Vegetation	1951

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	1565
Tree	13665
Water	8793
Runway	941
Urban	937
Vegetation	1661
Airfield	1010

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	1606
Tree	13686
Water	8793
Runway	994
Urban	876
Vegetation	1348
Airfield	807
Newbare	381

TABLE 6

May 1987

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	13808
Tree	7587
Water	7807

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	11129
Tree	9516
Water	8398
Runway	159

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	10484
Tree	9653
Water	8436
Runway	291
Urban	338

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	6969
Tree	10900
Water	8498
Runway	598
Urban	397
Vegetation	1528

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	6721
Tree	10861
Water	8491
Runway	552
Urban	416
Vegetation	1316
Airfield	333

Mixed Pixel, Least Squares Error = 5

CLASS	COUNT
Unclassified	6232
Tree	10906
Water	8493
Runway	600
Urban	421
Vegetation	1277
Airfield	304
Newbare	360

TABLE 7

May 1987

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	10541
Tree	10091
Water	8570

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	7171
Tree	12458
Water	8866
Runway	706

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	5712
Tree	12830
Water	8909
Runway	976
Urban	775

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	2980
Tree	12692
Water	8915
Runway	1119
Urban	717
Vegetation	2275

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	2336
Tree	12819
Water	8932
Runway	937
Urban	739
Vegetation	1768
Airfield	863

Mixed Pixel, Least Squares Error = 10

CLASS	COUNT
Unclassified	2308
Tree	12867
Water	8932
Runway	907
Urban	711
Vegetation	1614
Airfield	510
Newbare	480

TABLE 8

May 1987

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	9147
Tree	11304
Water	8751

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	5099
Tree	13526
Water	8901
Runway	1676

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	4018
Tree	13965
Water	8946
Runway	1090
Urban	1183

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	2043
Tree	13090
Water	8971
Runway	1226
Urban	924
Vegetation	2503

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	1782
Tree	13106
Water	8957
Runway	1107
Urban	846
Vegetation	2152
Airfield	662

Mixed Pixel, Least Squares Error = 15

CLASS	COUNT
Unclassified	1688
Tree	13210
Water	9015
Runway	983
Urban	818
Vegetation	1781
Airfield	422
Newbare	615

likelihood (S-classification), even with sigma = 3, cannot classify the inland body of water. The S-squared classifier makes that classification correctly, with sigma = 1.

The full results of the S-squared classifier are not easy to represent in image form because when a pixel is a mixture of several different classes, it is necessary to choose one of those classes (normally the one that accounts for most of the mixed pixel area) for the output display. **Plates 30 and 31** show, for T = 10, class maps corresponding to four different choices. These "lower-level" class maps display detail not seen on the original output. However, it should be emphasized again that the Complement Approach is designed to focus attention on the <u>unclassified</u> (black) part of the image.

The following legend is for Plates 6 thru 35

Color	Description
BLUE	Water
TAN	Vegetation/Trees
GREEN	Golf course grass (cultivated vegetation)
MAGENTA	Runway field (sparse vegetation)
YELLOW	Bare dirt
RED	Runway (Concrete)
CYAN	Mixed Urban
BLACK	Unclassified

1.2.4.3 Change detection using the S-squared classifier.

The ideal change detection algorithm for human activities should not signal as a change between dates something that has varied only as a result of natural processes (e.g., weather, season). **Plates 32 through 35** show the results of using the conventional S-classification for change detection between May 1985 and May 1987, compared with the new S-squared classifier.

The S-classification with sigma = 3 shows large areas that were not classified on either date, particularly in inland water, and other substantial areas that were classified in 1987 only.

The S-squared classification for T = 10 and sigma = 1 shows no such general pattern of change. Water in particular is classified correctly on both dates. In order to determine whether the indicated change pattern of the S-squared classification is a real one, ground truth would be needed. However, the S-squared method offers real promise as a change detection algorithm, simply because it is a more successful classifier of each date. The number of pixels that cannot be examined for change, because they were not classified on one date or the other, is very small.

1.3 Further Experiments

The number of available parameters in conventional spectral classification is already large. It includes the number of spectral bands, the choice of supervised or unsupervised classification, the choice of classification methods (maximum likelihood, Euclidean distance, parallelopipedon classifier, and others), the number of classes, and the variance (sigma-level) for pixel acceptance. To this list, the mixed-pixel spectral/spatial S-squared classifier adds options of its own: the number of spatial neighbors to be used, the number of classes permitted in neighboring pixels, and the tolerance to be set in accepting or rejecting a mixed pixel.

The current project was able to explore only a few of these alternatives. However, the results obtained suggest that, for at least one data set, mixed pixel S-squared analysis is very important. It is likely to prove more important yet, when the remaining unclassified pixels are examined in more detail, since it is these pixels which in the problem of small statistics are likely to yield the most interesting information. A conventional approach, which increases class variances to achieve higher percentages of classification, is likely to absorb those interesting unclassified pixels into existing classes; in other words, to throw away the pixels of most interest, merely in order to reduce the number of unclassified pixels of the scene (which are actually most likely to be mixed pixels of known classes, rather than representatives of new classes) to manageable proportions. The S-squared classifier, by identifying such mixed pixels from their spatial and spectral properties, removes them from consideration, without removing pixels that contain genuinely new and interesting information. This is illustrated for the two-band example case in **Figure 7**.

Before the above statement can be admitted as a firm conclusion, a number of additional experiments need to be performed. First, comparable runs need to be made with other data sets, preferably of a different geographical area and with ground truth available. Second, a number of the other parameters listed above need to be run through a range of values, rather than fixing them to one or two values as was necessary to keep the number of experiments on this project within bounds. The algorithm has now been through its own initial development variations, but there are still elements (e.g., mixed pixel tolerance) that

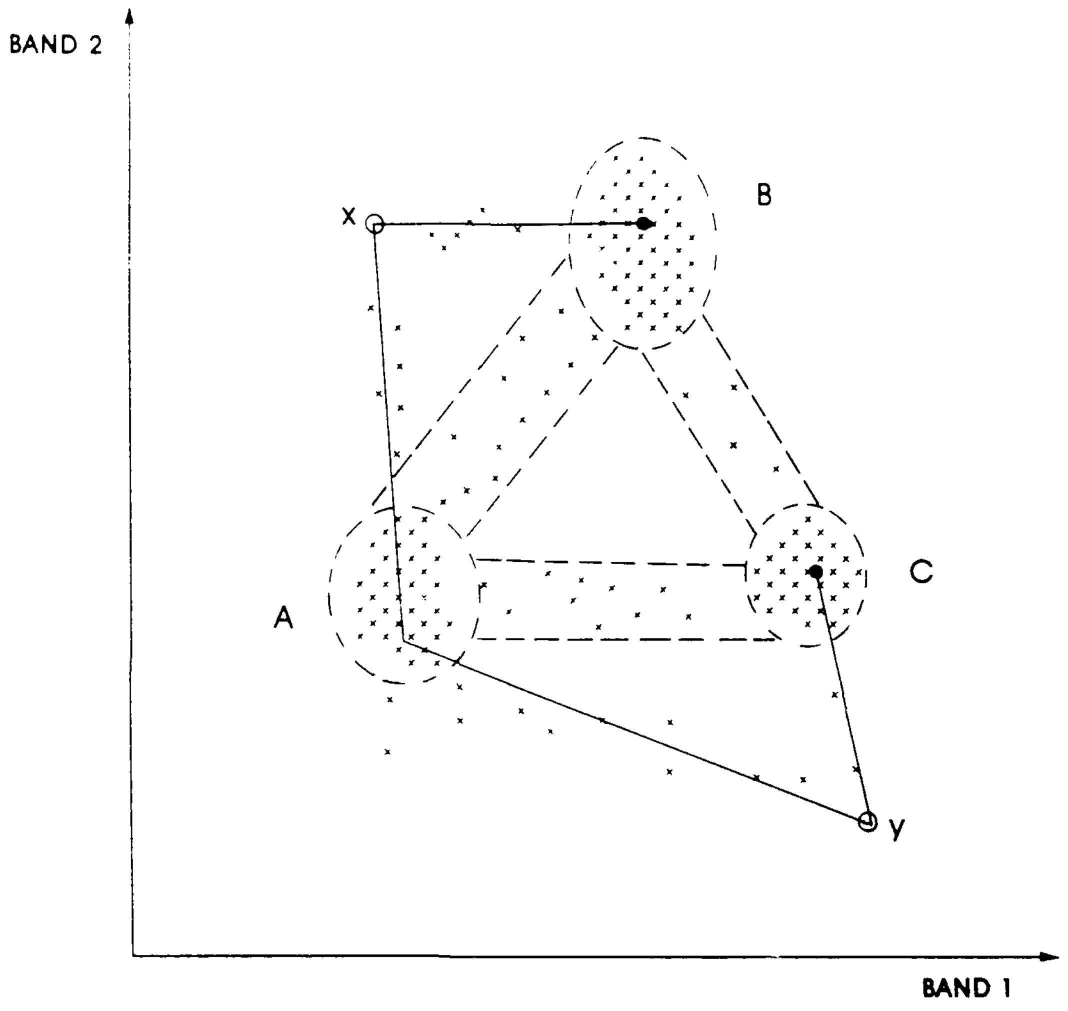


ILLUSTRATION SHOWS ONLY 2 BANDS FOR SIMPLICITY.

FIGURE 7: The detection of unknown classes

Pixels which cannot be classified as pure or mixed pixels from classes with known means and small variance are candidates to belong to new classes, or to be a combination of some new, unknown class and known classes. The unknown means of potential unknown classes are determined as the confluence points of vectors connecting existing class means and unclassified pixels.

need to be run over a range of values before final ones are adopted. Third, the software needs to be developed that seeks new materials at the partial pixel level, based only on the reflectances of pixels unclassified at the end of the existing S-squared classification.

The latter problem still needs additional analysis. As the General Approach of Section 1.1 makes clear, when we are dealing with complex mixed pixels, there is not enough information in a single pixel for an unknown material to be identified. Either a priori information must be provided (as to possible signatures) or some group of pixels must be identified, each of which is believed to possess some element of the unknown material. However, no experiments have so far been performed to test either of these alternatives in practice.

In addition, during the final classification runs of this project, another situation presented itself: there may be simple mixed pixels that happen to be entirely surrounded by pixels of some single other class. Such pixels, in the present algorithm, will never be classified correctly, even if all the necessary pure pixel classes have been identified. There are several possible ways to solve this problem, of which the simplest is to examine each unclassified pixel to see if it can be defined as a linear combination of the spectral signature of its neighbors and two known classes; however, no experiments have been performed to test how well such a process works.

1.4 Conclusions

1. Although the S-squared classifier has been run on only a small number of cases, it performs far better than a standard S-classifier (spectral only) in classifying scenes at a given sigma-level. Quantitatively, with sigma = 1 and T = 10, the classification success rate went in one case (1985) from 18 to 93 percent, and in another case (1987) from 23 to 92 percent.

2. If the above result is sustained for other scenes and other data types, the conventional view of the multispectral classification process should be changed. The most common concept in use today is that most pixels will lie within one of a few clusters in spectral space. These clusters, or classes, correspond to materials of a particular ground cover, i.e., they are "pure" clusters corresponding to a single surface cover such as water, forest, sand, etc. Geometrically, the picture is that of **Figure 5.d**. As mentioned earlier, the fact that the permitted sigma-level must be stretched to the statistically improbable value of 3 or 4 sigma in order to accommodate many pixels in a scene is a strong indication that there is something radically wrong with such an approach.

The S-squared classification results have shown just what is wrong. A different geometrical picture is needed. There are indeed clusters in spectral space, but these are small, tightly-defined clusters corresponding to a single, "pure", ground cover and having low variance. However, the majority of the pixels in the scene do not lie within these clusters. They lie instead in the lines, planes, and tetrahedra connecting the pairs, triplets, and quadruplets formed among the small, tightly-defined clusters. Geometrically, the correct picture is not that of **Figure 5.d**, it is that provided by **Figures 6 and 7**.

This new geometric insight is also the secret to solving the problem of small statistics. Expanding the size of clusters, as is done in conventional S-classification by increasing the permitted sigma-levels (variances), absorbs statistically insignificant but practically important pixels of unidentified ground cover into existing classes, where they are lost. The S-squared classification of the Complement Approach leaves precisely these unclassified pixels for further analysis, having removed all the well-defined pure pixels and simple mixed pixels from consideration. The S-squared approach is therefore recommended whenever the consideration of small statistics is important for successful scene analysis.

2. MULTISPECTRAL ANALYSIS USING THE FEATURE VECTOR APPROACH

2.1 Background

Since the first computer processing of multispectral images, dating back to aircraft scanners in the 1960's, linear combinations of bands have been explored to find "Feature Vectors" in multiple dimensional spectral space. Such vectors constitute directions, projected along which a particular type of scene ground cover is easily separated from the rest of the image content. The best known feature vector approaches are perhaps those of eigenvector (principal component) analysis, chlorophyll normal analysis, and tasseled cap analysis. Vectors have been sought to discriminate iron oxide and varnishes in geology, vegetation in agriculture, water depth in bathymetry, etc.

Analysis of images at EarthSat prior to 1988 often suggested the existence of a projective feature vector which discriminated man-made scene disturbances (features) from natural ground covers. Such a vector appeared to discriminate concrete, asphalt, building materials, and newly disturbed areas, from natural land cover and vegetation. In addition, differencing man-made feature vectors derived from multitemporal imagery appeared to be an effective means for identifying urban expansion. However, this feature vector was not scene independent. In this project, the possibility of deriving a stable (scene independent) feature vector was explored and the utility of the method for both scene analysis and change detection was evaluated.

2.2 Method of analysis

Five Thematic Mapper mosaics of parts of the Washington, D.C. area (see Plates 1-5) were employed in the analysis. From these scenes, a single composite image was created containing representative samples. A variety of linear combinations of the 6 TM bands (i.e., a variety of projection directions) were formed interactively, and evaluated for their ability to discriminate man-made features. The evaluation was performed subjectively by analysts familiar with the Washington, D.C. Metropolitan Area and environs. Initial results indicated that the best vectors were scene dependent. In other words, although useful vectors could be developed for each scene date, no single vector could be found which provided good discrimination on all images. The most likely cause of variation appeared to be changes in scene radiometry due to seasonal variations and atmospheric effects.

In an attempt to isolate and perhaps remove the factors causing date-to-date variation in the optimal feature vector, all imagery was normalized to the May 1985 image. The normalization was accomplished first by picking training sites which should be little affected by changes in vegetation or soil moisture (e.g., asphalt and concrete). A linear regression between training site statistics was then performed to derive a linear (gain and offset) look up table which could be used to adjust the image radiometry to match that of the May 1985 scene. These normalized images were then used in a second performance of the interactive linear combination analysis. Results indicated that the normalization did eliminate the effect of changes in solar illumination, detector calibration, and plant processing. However, the effects of atmospheric variation from image-to-image was not removed. The August 1985 image was particularly affected by atmosphere.

On the basis of these preliminary results, it was decided that the effects of the atmosphere, and also of intensity and vegetative vigor, needed to be removed prior to seeking a feature vector for human activities, if any scene independent result was to be obtained.

A step by step procedure was employed. (See Appendix 3 for a mathematical summary.) First, an intensity vector was derived from features such as asphalt and concrete. By projecting parallel to this vector, a set of five vectors remained (Thematic Mapper spectral space here was 6-dimensional since the thermal channel was not employed). In the remaining 5-space, a vegetation vector was then sought. This vector was identified using vegetated and non-vegetated features, and also features with varying degrees of vegetative vigor. By projection parallel to this vector, a 4-space was generated, orthogonal to both intensity and vegetation. Next, in the remaining 4-space, an atmosphere vector was identified by examination of areas showing varying degrees of atmospheric effect. Projection parallel to this vector reduced the remaining spectral space to one of three dimensions. In the final step, man-made and natural training areas were used in the residual 3-space to derive a feature vector discriminating man-made from natural ground covers. Of the directions orthogonal to this feature vector in the 3-space, one appears to correspond approximately to surface wetness, and the final vector has no identifiable physical correlation.

2.3 Results

The final vector coefficients for feature vectors corresponding to intensity, vegetation, atmosphere, man-made activities, wetness, and residual variation are shown in the following table:

TABLE II.1

Variable	Gain	Band 1	Band 2	Band 3	Band 4	Band 5	Band 7	Offset
Intensity	1.1	+.407	+.285	+.447	+.407	+.550	+.292	-57
Vegetation	1.9	-.193	-.137	-.217	+.913	-.222	-.119	+68
Atmosphere	4.6	+.777	-.077	-.569	+.000	+.062	-.252	-119
Man-Made	6.9	-.216	+.131	+.113	+.000	+.451	-.848	+165
Wetness	2.8	+.342	+.294	+.501	+.032	-.662	-.327	+99
Unnamed	23.5	-.174	+.889	-.408	+.000	-.053	+.099	+132

The images resulting from projection in the directions of vegetation, intensity, atmosphere, and man-made features are shown for each of the five processing dates in **Plate 36.**

The analysis of this project suggests that there is indeed a man-made feature vector which is independent of the main radiometric variations among different dates. This feature vector is a good discriminator of man-made land cover changes.

The value of the feature vector in change detection has so far proved inconclusive. There are other factors which affect the detection of man-made features. In particular, because many features are small, scene to scene registration may be required of sub-pixel accuracy.

2.4 Conclusions

On the basis of the experiments conducted here, 6-band Thematic Mapper data does permit the identification of a feature vector corresponding to human activities, which can be made independent of scene acquisition conditions. There appear to be two limiting factors to the value of this approach. The first is regional variability in building materials, which will cause changes to the preferred vector. We presently believe that building materials in North America are sufficiently standardized that a vector can be developed through the United States and Canada. New experiments would be needed to explore the effect of building materials in Europe and other continents.

Second, analysis of this type will be aided materially by higher spatial resolution (decreasing sensitivity to sub-pixel effects) and by additional spectral bands (which will offer a higher order subspace within which the man-made feature vector can be interactively sought).

As one interesting side observation, it should be noted that the vegetation vectors found here from Thematic Mapper imagery have negative coefficients in Band 5. This result appears contrary to vegetation/greenness vectors developed by other vectors.

APPENDIX 1

SOURCE CODE LISTING AND FLOW LOGIC FOR PROGRAMS DEVELOPED UNDER THIS CONTRACT

SOURCE CODE FOR MAXIMUM LIKELIHOOD SPECTRAL CLASSIFICATION

Include file for parm.h for statictical classifier

```
struct {
        int nclass;
        int nband;
        int bndnum[MAXBND];
        char clsnam[MAXCLS][32];
        int table[MAXBND][512];
        float incov[MAXCLS][MAXBND][MAXBND];
        float const[MAXCLS];
        float mean4[MAXCLS][MAXBND];
        float maxdist;
} parm;
```

<u>Main routine for statistical classifier</u>

```
#include <memory.h>
#include <malloc.h>
#include "estypes.h"
#include "esvalid.h"
#include "eserror.h"
#include "version.h"

#define   PARALLEL   0
#define   EUCLID     1
#define   MAXLIKE    2

main(argc, argv)
   int argc;
   char *argv[];
{
   int ihandle, ohandle;            /* Image handles */
   HEADER *inhdr, *outhdr;                 /* Image header pointers */
   Pixel **buff;                    /* Pointer to input buffers */
   Pixel *obuf;                     /* Pointer to output buffer */
   int row1, nrows;                 /* 1st row, # of rows */
   int col1, ncols;                 /* 1st col, # of cols */
   unsigned
   int nclass,                      /* # of classes */
       ntable,                             /* # of tables required */
       cmbands;                     /* # of common bands */
   int imbands;                     /* # of images bands */
   int *iband;                             /* Pointer to image bands */
   int stbands;                     /* # of bands in stats file */
   int *sband;                             /* Pointer to stats bands */
   int *cband;                             /* Pointer to common bands */
   int *min, *max;                  /* Class mins/maxes */
   char *name;                             /* Class names */
   int *count;                             /* Class counts */
   int *number;                     /* Class numbers */
   float *mean;                     /* Class mean vectors */
   float *var;                             /* Class covariance matrices */
   int *table;                             /* Parallelepiped tables */
   float *ivar;                     /* Inverted covar matrices */
   float *lndet;                    /* Constant portion of max
                                       likelihood equation */
   char iname[129], oname[129];     /* Image names */
   char sname[129];                        /* Stats file name */
   char lname[129];                 /* List file name */
   FILE *fp,                        /* Stats file pointer */
```

```
        *lp;                          /* List file pointer */
    int method;                              /* Classification method */
    float *maxdist;                   /* Max allowable distance */
    int class, maxclass;
    int band, row;
    int total, unclass, device;
    float xnum;
    char *calloc();
    static char *menu[4] = {"Parallelepiped",
                            "Euclidean distance",
                            "Maximum Likelihood",
                            NULL};

    title(argc, argv, "Supervised classification", VMAJOR, VMINOR);
/*
 *  Open input image
 */
    while (TRUE)
    {
        iname[0] = 0;
        ihandle = open_image("Enter input image name", iname, "img",
                             IO_READ, -1);
        if (ihandle < 0)
            goto done;
    /*
     *  Make sure there's more than 1 band in image
     */
        iband = get_bands(ihandle, &imbands);
        if (iband == NULL)
            goto done;
        if (imbands >1)
            break;
        put_string("Selected image has only one band\n\n");
        close_image(ihandle);
    }
/*
 *  Get processing window
 */
    clear_screen();
    if (select_wind(ihandle, "Select Processing Window", &row1, &nrows,
                              &col1, &ncols, 1))
        goto done;
/*
 *  Open stats file.
```

Main routine for statistical classifier

```
 */
   clear_screen();
   new_ext(iname, "stats", sname);
   while (TRUE)
   {
       fp = open_text("Enter stats file name", sname, "stats",
                      IO_READ);
       if (fp == NULL)
           goto done;
       if (read_stats(fp, &stbands, &sband, &nclass, &number, &name,
                      &count, &min, &max, &mean, &var))
       {
           (void)fclose(fp);
           sname[0] = 0;
           continue;
       }
/*
 *  Clear out the class counters
 */
   (void)memset((char *)count, 0, (int)nclass*sizeof(int));
/*
 *  Get list of common bands
 */
       blist(iband, imbands, sband, stbands, &cband, &cmbands);
       if (cmbands > 1)
           break;
       put_string("Image and stats have < 2 bands in common\n\n");
       free((char *)var);
       free((char *)mean);
       free((char *)max);
       free((char *)min);
       free((char *)count);
       free((char *)name);
       free((char *)number);
       free((char *)sband);
       free((char *)cband);
       (void)fclose(fp);
   }
/*
 *  Prompt for bands to use
 */
   if (which_bands(cband, &cmbands))
       goto done;
/*
```

Main routine for statistical classifier

```
 *  Compress stats to eliminate any unused bands
 */
   comp_stats(sband, stbands, cband, cmbands, nclass, mean, var);
/*
 *  Get classification method
 */
   clear_screen();
   if (ask_menu("SELECT CLASSIFICATION METHOD", menu, &method, 0))
       goto done;
   method--;          /* Make 0 relative */

   put_string("\n\n\n");
/*
 * Get max allowable distance for each class.
 */
   maxdist = (float *)malloc(nclass*sizeof(float));
   if (maxdist == NULL)
       goto badalloc;

   clear_screen();
   if (method == MAXLIKE)
   {
       put_string("\n\nNOTE: Maximum distance will be normalized by\n");
       put_string("sqrt(nbands) so that a distance of 1 will yield\n");
       put_string("approximately 66%% of training pixels classified.\n\n");
   }
   if (get_dist(nclass, cmbands, var, maxdist, name))
       goto done;
/*
 *  Make parallelepiped tables
 */
   ntable = (nclass + 31)/32;
   table = (int *)calloc(ntable*256*cmbands, sizeof(int));
   if (table == NULL)
       goto badalloc;

   make_table(method, table, nclass, cmbands, maxdist, mean, var);
/*
 *  If maximum likelihood, set up inverted covariance matrices
 *  and compute natural log of the determinant of the covariance
 *  matrix for each class
 */
   if (method == MAXLIKE)
   {
        lndet = (float *)malloc(nclass*sizeof(float));
        if (lndet == NULL)
            goto badalloc;
        ivar = (float *)malloc(nclass*cmbands*cmbands*sizeof(float));
        if (ivar == NULL)
            goto badalloc;
        if (setup_max(nclass, cmbands, var, lndet, ivar, number))
            goto done;
    }
    device = 3;
    clear_screen();
    if (ask_list("\nSelect device for summary listing",
                "terminal,file,both", &device, 1))
        goto done;
    if (device > 1)
    {
        new_ext(iname, "lis", lname);
        lp = open_text("\nEnter name for list file", lname, "lis",
                        IO_WRITE);
        if (lp == NULL)
            goto done;
    }
/*
 *  Open output image
 */
    oname[0] = 0;
    ohandle = open_image("Enter output image name", oname, "img",
                        IO_WRITE, -1);
    if (ohandle < 0)
        goto done;
    inhdr = get_header(ihandle, 0);
    if (inhdr == NULL)
        goto done;
    outhdr = get_header(ohandle, 1);
    if (outhdr == NULL)
        goto done;
    wndw_header(outhdr, inhdr, row1, nrows, col1, ncols, 1, 1);
    outhdr->num_bands = 1;
    outhdr->band[0].band_num = 1;
/*
 *  Determine maximum class number
 */
    for (class = maxclass = 0; class < nclass; class++)
        if (maxclass < number[class])
```

```
            maxclass = number[class];
        outhdr->maxclass = maxclass;
        if (write_header(ohandle))
            goto done;
/*
 *   Allocate space for input buffer pointers
 */
        buff = (Pixel **)malloc(cmbands*sizeof(char **));
        if (buff == NULL)
            goto badalloc;
/*
 *   Start classifying
 */
        if (set_bands(ihandle, cmbands, cband))
            goto done;
        if (set_bands(ohandle, 1, &outhdr->band[0].band_num))
            goto done;
/*
 *   Process each line
 */
        put_string("\n");
        for (row = row1; row < (row1 + nrows); row++)
        {
            count_down(row, row1, nrows);
/*
 *   Read each band
 */
            for (band = 0; band < cmbands; band++)
            {
                buff[band] = get_image(ihandle, cband[band], row);
                if (buff[band] == NULL)
                    goto done;
                buff[band] += col1 - inhdr->col_start;
            }
/*
 *   Get output buffer
 */
            obuf = put_image(ohandle, 1, row);
            if (obuf == NULL)
                goto done;
/*
 *   Call appropriate subroutine
 */
            switch (method)
        {
        case PARALLEL:
            parallel(buff, obuf, ncols, cmbands, table, ntable,
                        number, count);
            break;
        case EUCLID:
            euclid(buff, obuf, ncols, cmbands, table, ntable, number,
                      mean, count, maxdist);
            break;
        case MAXLIKE:
            maxlike(buff, obuf, ncols, cmbands, table, ntable, number,
                       mean, ivar, lndet, count, maxdist);
            break;
        }
    }
    put_string("\n");
/*
 *  List stats
 */
    for (class = total = 0; class < nclass; class++)
        total += count[class];
    unclass = nrows*ncols - total;
    xnum = (float)(nrows*ncols)/100.;
    clear_screen();
    print_list(device, lp,
            "Number  Name                 Count    Percent\n");
    print_list(device, lp, "%4d    Unclassified%12d%11.2f\n",
            0, unclass, (float)unclass/xnum);
    for (class = 0; class < nclass; class++)
    {
        print_list(device, lp, "%4d    %-16s%8d%11.2f\n", class+1, name,
                   count[class], (float)count[class]/xnum);
        name += 17;
    }
    if (device & 1)
        (void) pause();
done:
    close_image(IO_ALL);
    exit(ioerror());
badalloc:
    set_error(E_MALLOC);
    goto done;
}
```

Function which_bands for statistical classifier

```
#include "esvalid.h"
#include <stdio.h>
#include <malloc.h>

int which_bands(list, nband)
/*
 *  Choose bands from list of common bands
 */
  int         *list;          /* List of common bands */
  int         *nband;               /* Number of common bands */
{
  int         *tlist,         /* Pointer to temp list */
              ntouse,
               min = 2;
  char                cbuf[81],
              *allocate();
/*
 *  If only two common bands, return
 */
  if (*nband < 3)
    return (0);
/*
 *  Allocate temporary list and copy the list
 */
  tlist = (int *) allocate(*nband * sizeof(int));
  if (tlist == NULL)
    return (1);

  (void) memcpy(tlist, list, *nband*sizeof(int));
/*
 *  Build prompt
 */
  clear_screen();
  num_list(list, *nband, cbuf, 81);
  put_int(*nband, "\n\nImage and stats have %d bands in common\n");
  put_string("Bands are numbered ");
  put_string(cbuf);
  put_string("\n\n");

  (void) sprintf(cbuf,
                "Enter the number of bands to be processed (%d-%d)",
                2, *nband);

  min = 2;
```

Function which_bands for statistical classifier

```
ntouse = *nband;
if (ask_int(cbuf, &ntouse, 1, 1, V_GEMIN|V_LEMAX, &min, nband))
      return (1);

if (*nband > ntouse)
{
   if (ask_int("\nEnter band numbers to use", list, ntouse, 0,
               -(*nband), tlist, NULL))
      return (1);
   *nband = ntouse;
}
isort(list, ntouse, 0);
return (0)ndx
```

Function make_table for statistical classifier

```
#include <math.h>

#define  EUCLID  1
#define  MAXLIKE  2

void make_table(method, table, nclass, nband, nstdv, mean, var)
/*
 *  Make the parallelepiped tables. Each table has a 256 element
 *  lookup for each band which describes the parallelepipeds for
 *  32 classes. The 1st table describes classes 1-32 with class
 *  1 corresponding to the sign bit of each table entry and class
 *  32 corresponding to the low order bit. The bits are set
 *  according to whether the parallelepiped for a class contains
 *  the corresponding grey level for that band.
 */
   int  method;
   int *table;                    /* Pointer to the tables */
   int nclass;               /* Number of classes */
   int nband;                     /* Number of bands */
   float *nstdv;             /* Size of ||epiped in terms
                               of # of std deviations about mean */
   float *mean;              /* Mean vectors */
   float *var;                    /* Covariance matrices */
{
   unsigned int mask;             /* Bit mask for class */
   int band, class, grey;
   int min, max;
   int *tptr;                /* Pointer to current table set */
   float size;
   float maxstdv, stdv, *vptr;

   mask = 0x80000000;

   for (class = 0; class < nclass; class++)
   {
      tptr = table;
/*
 * For Euclidean distance, set all variances to the max band variance.
 */
      switch (method)
      {
      case EUCLID:
         maxstdv = 0.;
         vptr = var;
```

```
        for (band = 0; band < nband; band++)
        {
            if (maxstdv < vptr[band])
                maxstdv = vptr[band];
            vptr += band + 1;
        }
        maxstdv = sqrt(maxstdv);
        vptr = var;
        for (band = 0; band < nband; band++)
        {
            vptr[band] = maxstdv;
            vptr += band + 1;
        }
        break;
    case MAXLIKE:
        nstdv[class] *= sqrt((double) nband);
        break;
    }
    for (band = 0; band < nband; band++)
    {
        stdv = sqrt(var[band]);
        size = (int)(nstdv[class]*stdv + 0.5);
        min = (int)(mean[band] - size);
        if (min < 0) min = 0;
        max = (int)(mean[band] + size);
        if (max > 255) max = 255;
        for (grey = min; grey <= max; grey++)
            tptr[grey] |= mask;
        var += band + 1;        /* Point to next line of matrix */
        tptr += 256;      /* Point to next band of table */
    }
    mean += nband;              /* Point to next mean vector */
    mask >>= 1;                 /* Update mask, if 0 then reset */
    if (!mask)
    {
        mask = 0x80000000;
        table += nband*256;
    }
    switch (method)
    {
    case EUCLID:        /* Break intentionally omitted */
        nstdv[class] *= maxstdv;
    case MAXLIKE:
        nstdv[class] *= nstdv[class];
```

```
            break;
         }
      }
   }
```

Function parallel for statistical classifier

```
#include "estypes.h"

void parallel(buff,obuf,ncols,nbands,table,ntable,number,count)
/*
 *   parallel - Classify a single line using a parallelepiped
 *            algorithm
 */
   Pixel *buff[];           /* Input buffer pointers */
   Pixel obuf[];            /* Output buffer */
   int ncols;               /* Number of columns per line */
   int nbands;                   /* Number of image bands */
   int *table;                   /* Pointer to ||epiped tables */
   int ntable;                   /* Number of ||epiped tables */
   int number[];            /* Class numbers */
   int count[];             /* Pointer to class counters */
{
   int pxl;                 /* Pixel index */
   int pixel;               /* Pixel value */
   int band;
   unsigned int class;
   unsigned int tbl;        /* Bit table base index */
   union {                  /* Parallelepiped bit mask */
      unsigned int umask; /* unsigned for logical shifting */
      int mask;             /* signed for testing sign bit */
   } bitmask;
   int *curtab;             /* Pointer to current table */
/*
 *   Each bit table contains info for 32 classes with one bit
 *   assigned to each class. The pixel grey levels for each band are
 *   used to index and select a bit mask for each band. When these are
 *   all logically ANDed together any bits which remain set represent
 *   classes to which the pixel may belong.
 */
/*
 *   This loop classifies one pixel per pass
 */
   for (pxl = 0; pxl < ncols; pxl++) {
/*
 *   This loop cycles through all bit tables to perform parallelepiped
 *   classification
 */
      curtab = table;
      for (tbl = 0; tbl < ntable; tbl++) {
         class = tbl << 5;              /* Base class # for table */
```

```
            pixel = buff[0][pxl];
            bitmask.umask = curtab[pixel];
            curtab += 256;
/*
 *   AND all bit masks together
 */
            for (band = 1; band < nbands; band++) {
                pixel = buff[band][pxl];
                bitmask.umask &= curtab[pixel];
                curtab += 256;
            }
/*
 *   Look for bits left on.
 */
            while (bitmask.umask) {
                if (bitmask.mask < 0) {
/*
 *   Sign bit is on. This is the class
 *   Store class value in output buffer and increment
 *   histogram counter
 */
                    obuf[pxl] = (Pixel)number[class];
                    count[class]++;
                    goto nextpix;
                }             /* End of if (bitmask.mask < 0) */
/*
 *   Shift next bit into sign position. Use unsigned
 *   version to insure a logical rather than arithmetic
 *   shift.
 */
                bitmask.umask <<= 1;
                class++;              /* Increment class number */
            }         /* End of while (bitmask.umask) */
        }             /* End of for (tbl .... */
        obuf[pxl] = 0;
nextpix:
        continue;
    }         /* End of for (pxl = .... */
}
```

Function euclid for statistical classifier

```
#include <malloc.h>
#include "estypes.h"
#include "eserror.h"

void euclid(buff,obuf,ncols,nbands,table,ntable,number,mean,
        count,maxdist)
/*
 *   euclid - Classify a single line using a hybrid parallelepiped/
 *   Euclidean distance algorithm
 */
    Pixel *buff[];              /* Input buffer pointers */
    Pixel obuf[];               /* Output buffer */
    int ncols;                  /* Number of columns per line */
    int nbands;                     /* Number of image bands */
    int *table;                     /* Pointer to ||epiped tables */
    int ntable;                     /* Number of ||epiped tables */
    int number[];               /* Class numbers */
    float *mean;                /* Pointer to class means */
    int count[];                /* Pointer to class counters */
    float *maxdist;             /* Maximum euclidean distance */
{
    int pxl;                    /* Pixel index */
    int band;
    unsigned int class;
    unsigned int tbl;           /* Bit table base index */
    union {                     /* Parallelepiped bit mask */
        unsigned int umask; /* unsigned for logical shifting */
        int mask;               /* signed for testing sign bit */
    } bitmask;
    int best;                   /* Current "best" class */
    static int *pixel = NULL;       /* int version of pixel values */
    static float *rpix;         /* float version of pixel values */
    float mindst;               /* current minimum distance */
    float dist;                 /* Euclidean distance from class mean */
    float diff;
    int *curtab;                /* Pointer to current table */
    float *curmean;             /* Pointer to current mean vector */
/*
 *  On the 1st pass, allocate memory for pixel vectors
 */
    if (pixel == NULL) {
        pixel = (int *)malloc((unsigned)nbands*sizeof(int));
        if (pixel == NULL)
            goto badalloc;
        rpix = (float *)malloc((unsigned)nbands*sizeof(float));
        if (rpix == NULL)
            goto badalloc;
    }
/*
 *   Each bit table contains info for 32 classes with one bit
 *   assigned to each class. The pixel grey levels for each band are
 *   used to index and select a bit mask for each band. When these are
 *   all logically ANDed together any bits which remain set represent
 *   classes to which the pixel may belong.
 */
/*
 *   This loop classifies one pixel per pass
 */
    for (pxl = 0; pxl < ncols; pxl++) {
/*
 *   Convert pixel values to int and float
 */
        for (band = 0; band < nbands; band++) {
            pixel[band] = buff[band][pxl];
            rpix[band] = pixel[band];
        }
/*
 *   Initialize to no class found.
 */
        best = -1;
        mindst = 9999999.;
/*
 *   This loop cycles through all bit tables to perform parallelepiped
 *   classification
 */
        curtab = table;
        for (tbl = 0; tbl < ntable; tbl++) {
            class = tbl << 5;              /* Base class # for table */
            bitmask.umask = curtab[pixel[0]];
            curtab += 256;
/*
 *   AND all bit masks together
 */
            for (band = 1; band < nbands; band++) {
                bitmask.umask &= curtab[pixel[band]];
                curtab += 256;
            }
/*
```

<u>Function euclid for statistical classifier</u>

```
 *   Look for bits left on.
 */
          while (bitmask.umask) {
              if (bitmask.mask < 0) {
/*
 *   Bit is on. Compute Euclidean distance
 *   and compare to current minimum.
 */
                  curmean = mean + class*nbands;
                  diff = rpix[0] - curmean[0];
                  dist = diff*diff;
                  for (band = 1; band < nbands; band++) {
                      diff = rpix[band] - curmean[band];
                      dist += diff*diff;
                  }
/*
 *   Compare to max allowable Euclidean distance
 */
                  if (dist <= maxdist[class]) {
/*
 *   Compare to current min.
 */
                      if (dist < mindst) {
                          mindst = dist;
                          best = class;
                      }
                  }
              }               /* End of if (bitmask.mask < 0) */
/*
 *   Shift next bit into sign position. Use unsigned
 *   version to insure a logical rather than arithmetic
 *   shift.
 */
              bitmask.umask <<= 1;
              class++;              /* Increment class number */
          }           /* End of while (bitmask.umask) */
      }               /* End of for (tbl .... */
/*
 *   Store class value in output buffer and increment
 *   histogram counter
 */
      if (best >= 0) {
          obuf[pxl] = number[best];
          count[best]++;
```

```
            }
            else
                obuf[pxl] = 0;
        }           /* End of for (pxl = .... */
        return;
    badalloc:
        set_error(E_MALLOC);
        exit(ioerror());
    }
```

Function setup_max for statistical classifier

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "eserror.h"

int setup_max(nclass, nbands, var, const, ivar, number)
/*
 *  Setup for maximum likelihood classification.
 *    1) Convert covariance matrices from lower triangle
 *       storage to rectangular storage and invert them.
 *    2) Compute constant part of probability equation for
 *       each class. That is log(determinant)
 */
    int nclass;               /* Number of classes */
    int nbands;                      /* Number of bands */
    float *var;                      /* Covariance matrices (lower triangle */
    float const[];            /* Constants */
    float *ivar;              /* Inverted covariance matrices
                                 (stored in rectangular form) */
    int *number;              /* Class numbers */
{
    int class, band, band2;
    double *rvar;             /* Rectangular version of covar */
    double determ;            /* Determinant of covariance matrix */
    double *vv;                      /* Temp storage */
    int *indx;                /* Temp storage */
    int idx, icode, nelem;

    nelem = nbands*(nbands + 1)/2;
/*
 *  Allocate some temp storage
 */
    rvar = (double *)malloc((unsigned)nbands*nbands*sizeof(double));
    if (rvar == NULL)
        goto badalloc;

    vv = (double *)malloc((unsigned)nbands*sizeof(double));
    if (vv == NULL)
        goto badalloc;

    indx = (int *)malloc((unsigned)nbands*sizeof(int));
    if (indx == NULL)
        goto badalloc;
```

Function setup_max for statistical classifier

```
    for (class = 0; class < nclass; class++, var += nelem)
    {
/*
 *  Convert covar to double rectangular
 */
        for (band = idx = 0; band < nbands; band++)
        {
            for (band2 = 0; band2 < band; band2++)
            {
                rvar[band*nbands+band2] = rvar[band2*nbands+band] =
                        var[idx++];
            }
            if ( var[idx] <= 0.0 )
                var[idx] = 1.0;
            rvar[band*(nbands+1)] = var[idx++];
        }
/*
 *  Compute determinant
 */
        mdet_(rvar, &nbands, &determ, indx, vv, &icode);
        if (icode)
        {
            set_error(E_LOGIC);
            exit(ioerror());
        }
/*
 *  Compute log of determinant
 */
        if ( determ <= 0.0 )
        {
            put_string( "Error in class %d covar matrix\n",
                        number[class] );
            put_string( "Determinant = %f\n", determ );
            put_string( "Determinant reset to 1e-10\n" );
            determ = 1e-10;
        }
        const[class] = log(determ);
/*
 *  Convert covar to double rectangular again because it was
 *  destroyed by mdet.
 */
        for (band = idx = 0; band < nbands; band++)
        {
            for (band2 = 0; band2 < band; band2++)
```

```
          {
             rvar[band*nbands+band2] = rvar[band2*nbands+band] =
                   var[idx++];
          }
          rvar[band*(nbands+1)] = var[idx++];
       }
/*
 *  Invert the covariance matrix
 */
       matinv_(rvar, &nbands, indx, &icode);
       if (icode)
       {
          put_error("Singular matrix (setup_max)");
          set_error(E_LOGIC);
          exit(ioerror());
       }
/*
 *  Transfer to output matrix in single precision form
 */
       for (idx = 0; idx < nbands*nbands; idx++)
          ivar[idx] = rvar[idx];
       ivar += idx;
    }         /* End of class loop */
    free((char *)indx);
    free((char *)vv);
    free((char *)rvar);
    return (0);
badalloc:
    set_error(E_MALLOC);
    return (1);
}
```

Function maxlike for statistical classifier

```
#include <malloc.h>
#include "estypes.h"
#include "eserror.h"

void maxlike(buff, obuf, ncols, nbands, table, ntable, number, mean,
            ivar, const, count, maxdist)
/*
 *   maxlike - Classify a single line using a hybrid parallelepiped/
 *   maximum likelihood algorithm
 */
   Pixel      *buff[],        /* Input buffer pointers */
              *obuf;               /* Output buffer */
   int         ncols,              /* Number of columns per line */
               nbands,        /* Number of image bands */
              *table,              /* Pointer to ||epiped tables */
               ntable,        /* Number of ||epiped tables */
              *number,        /* Class numbers */
              *count;              /* Pointer to class counters */
   float      *mean,               /* Pointer to class means */
              *ivar,          /* Pointer to inverted covar matrices */
              *const,              /* Pointer to class constants */
              *maxdist;       /* Maximum normalized distance */
{
   int pxl;                   /* Pixel index */
   int band;
   unsigned int class;
   unsigned int tbl;          /* Bit table base index */
   int row, col;
   union
   {                          /* Parallelepiped bit mask */
      unsigned int umask; /* unsigned for logical shifting */
      int mask;               /* signed for testing sign bit */
   } bitmask;
   int best;                  /* Current "best" class */
   static int *pixel = NULL;       /* int version of pixel values */
   static float *rpix;        /* float version of pixel values */
   static float *diff;        /* pixel values minus band means */
   float minprob;             /* current min negative probability */
   float dist;                /* normalized distance from mean */
   float sum, prob;
   int *curtab;               /* Pointer to current table */
   float *curmean;            /* Pointer to current mean vector */
   float *curvar;             /* Pointer to current covar matrix */
   float *cvar;               /* Temp covar pointer */
```

Function maxlike for statistical classifier

```
/*
 *  On the 1st pass, allocate memory for pixel vectors
 */
   if (pixel == NULL)
   {
       pixel = (int *)malloc((unsigned)nbands*sizeof(int));
       if (pixel == NULL)
           goto badalloc;
       rpix = (float *)malloc((unsigned)nbands*sizeof(float));
       if (rpix == NULL)
           goto badalloc;
       diff = (float *)malloc((unsigned)nbands*sizeof(float));
       if (diff == NULL)
           goto badalloc;
   }
/*
 *   Each bit table contains info for 32 classes with one bit
 *   assigned to each class. The pixel grey levels for each band are
 *   used to index and select a bit mask for each band. When these are
 *   all logically ANDed together any bits which remain set represent
 *   classes to which the pixel may belong.
 */
/*
 *   This loop classifies one pixel per pass
 */
   for (pxl = 0; pxl < ncols; pxl++)
   {
/*
 *   Convert pixel values to int and float
 */
       for (band = 0; band < nbands; band++)
       {
           pixel[band] = buff[band][pxl];
           rpix[band] = pixel[band];
       }
/*
 *   Initialize to no class found.
 */
       best = -1;
       minprob = 999999999.;
/*
 *   This loop cycles through all bit tables to perform parallelepiped
 *   classification
 */
```

```
      curtab = table;
      for (tbl = 0; tbl < ntable; tbl++)
      {
         class = tbl << 5;              /* Base class number for table */
         bitmask.umask = curtab[pixel[0]];
         curtab += 256;
/*
 *    AND all bit masks together
 */
         for (band = 1; band < nbands; band++)
         {
            bitmask.umask &= curtab[pixel[band]];
            curtab += 256;
         }
/*
 *    Look for bits left on.
 */
         while (bitmask.umask)
         {
            if (bitmask.mask < 0)
            {
/*
 *    Bit is on. Compute multivariate probability
 *    and compare to current minimum. First compute
 *    difference vector from pixel to class mean
 */
               curmean = mean + class*nbands;
               curvar = ivar + class*nbands*nbands;
               for (band = 0; band < nbands; band++)
                  diff[band] = rpix[band] - curmean[band];
/*
 *    Now compute transpose of difference vector times
 *    inverted class covariance matrix times difference
 *    vector. Result is a scalar.
 */
               dist = 0.;
               for (row = 0; row < nbands; row++)
               {
                  cvar = curvar;
                  sum = diff[0]*cvar[0];
                  for (col = 1; col < nbands; col++)
                  {
                     cvar += nbands;        /* Next column */
                     sum += diff[col]*cvar[0];
```

```
                    }
                    dist += sum*diff[row];
                    curvar++;    /* Point to next covar row */
                }
/*
 *   Compare to max allowable normalized distance
 */
                if (dist <= maxdist[class])
                {
/*
 *   OK, compute minus twice the natural log of multivariate
 *   probability and compare to current min.
 *   (Forget the 2*pi **n/2 part)
 */
                    prob = const[class] + dist;
                    if (prob < minprob)
                    {
                        minprob = prob;
                        best = class;
                    }
                }
            }              /* End of if (bitmask.mask < 0) */
/*
 *   Shift next bit into sign position. Use unsigned
 *   version to insure a logical rather than arithmetic
 *   shift.
 */
            bitmask.umask <<= 1;
            class++;              /* Increment class number */
        }          /* End of while (bitmask.umask) */
      }            /* End of for (tbl .... */
/*
 *   Store class value in output buffer and increment
 *   histogram counter
 */
      if (best >= 0)
      {
         obuf[pxl] = number[best];
         count[best]++;
      }
      else
         obuf[pxl] = 0;
   }          /* End of for (pxl = .... */
   return;
```

```
badalloc:
   set_error(E_MALLOC);
   exit(ioerror());
}
```

Function get_dist for statistical classifier

```
#include <math.h>
#include <stdio.h>
#include "esvalid.h"

int get_dist(nclass, nband, var, maxdist , name)
/*
 * Get max distance for classification in terms of # of std
 * deviations about the mean. Allow user to enter a single
 * distance to use for all classes or to enter distances
 * individually by class
 */
    int         nclass,       /* # of classes */
                nband;                /* # of image bands */
    float      *var,          /* Covar matrices (lower triangle) */
               *maxdist;      /* Returned max dist */
    char        name [][17]; /* Class names */
{
    int         individ,      /* Get individual distances ? */
                ciass,        /* Class index */
                band,                 /* Band index */
                nelem;                /* # elements per cov matrix */
    float      *vptr,         /* Pointer into covariance */
                min = 0.0;

    if (ask_query("Specify maximum classification distances by class",
                &individ, 0))
        return(1);

    if (individ)
    {
        nelem = (nband + 1)*nband/2;
        for (class = 0; class < nclass; class++)
        {
            vptr = &var[class*nelem];
/*
 * Print class band std deviations
 */
            put_string("\nBand std. deviations for class %d (%s):",
                    class+1, name[class]);
            for (band = 0; band < nband; band++)
            {
                put_string("  %.1f", sqrt(vptr[band]));
                vptr += band + 1;
            }
            put_string("\n");
            maxdist[class] = (class) ? maxdist[class-1] : 3.0;
            if (ask_float(
                "Enter max dist in std. deviations for this class",
                &maxdist[class], 1, 1, V_GTMIN, &min, (float *)NULL))
                return(1);
        }
    }
    else
    {
        maxdist[0] = 3.0;
        if (ask_float(
            "Enter max dist in std. deviations for all classes",
            maxdist, 1, 1, V_GTMIN, &min, (float *)NULL))
            return(1);
        for (class = 1; class < nclass; class++)
            maxdist[class] = maxdist[0];
    }
    return(0);
}
```

SOURCE CODE FOR MIXED PIXEL CLASSIFICATION

Include file defs.h for mixed pixel classifier

```
#define     MAXCLS     4
#define     MAXNBR     4
#define     NULL_CLASS 255

#define  max(x,y)   (((x) > (y)) ? (x) : (y))
#define  min(x,y)   (((x) < (y)) ? (x) : (y))

extern int          iterate(),
                    lsfit(),
                    clsline();
```

Main routine for mixed pixel classifier

```
#include "estypes.h"
#include "eserror.h"
#include "esvalid.h"
#include "defs.h"
#include "version.h"
#include <math.h>

main(argc, argv)
/*
 * This program is a "mixed pixel" locator. It takes as input
 *    1) A multi-band image
 *    2) A single or multi-band class map.
 *    3) A statistics file containing info about each class in the
 *       class map. Only the mean vector is used from this file.
 *
 * The program attempts to classify previously unclassified pixels
 * which are adjacent (using 4-way connectivity) to classified
 * pixels by computing linear weights of the surrounding class means
 * which could account for the observed spectral values of the pixel.
 */
   int          argc;
   char        *argv[];
{
   HEADER        *ihdr, *chdr;
   FILE        *fp;
   int          ihndl, chndl,
                nbandi, nbandc, n, m,
                nbands, nclass, *cnum, *count,
                roff, coff, isect, *sband, *iband,
                match, row1, nrows, col1, ncols,
               *cmbnd, ncomm, iter, nchange,
               *clsbnd, *mins, *maxs;
   float       *mean, *var, maxerr, minwgt, zero = 0.0;
       double alpha_mean;
       double alpha_sd;
       struct
       {
               double sum;
               double sumsq;
               double minalpha;
               double maxalpha;
       } stats;    /* Statistics for the linear weights */
   char         iname[129], cname[129], *clsnm,
               *allocate();
```

Main routine for mixed pixel classifier

```
    Pixel       *buff, *clsbuf[3][MAXCLS], **imgbuf;

    title(argc, argv, "Mixed pixel classifier", VMAJOR, VMINOR);

    iname[0] = 0;
    ihndl = open_image("Enter image name", iname, "img", IO_READ, 0);
    if (ihndl < 0)
        goto done;

    ihdr = get_header(ihndl, 0);
    if (ihdr == NULL)
        goto done;

    iband = get_bands(ihndl, &nbandi);
    if (nbandi < MAXCLS)
    {
        put_string("Image MUST have %d or more bands\n", MAXCLS);
        set_error(E_LOGIC);
        goto done;
    }

    new_ext(iname, "stats", iname);
    fp = open_text("Enter stats file name", iname, "stats", IO_READ);
    if (fp == NULL)
        goto done;

    if (read_stats(fp, &nbands, &sband, &nclass, &cnum, &clsnm, &count,
                  &mins, &maxs, &mean, &var))
        goto done;

    (void) fclose(fp);
/*
 * Get list of common bands
 */
    blist(iband, nbandi, sband, nbands, &cmbnd, &ncomm);
    if (ncomm < MAXCLS)
    {
        put_string("Image and stats have < %d common bands\n", MAXCLS);
        set_error(E_LOGIC);
        goto done;
    }
/*
 * Compress out unwanted bands
 */
```

Main routine for mixed pixel classifier

```
    comp_stats(sband, nbands, cmbnd, ncomm, nclass, mean, var);

    cname[0] = 0;
    chndl = open_image("Enter class map name", cname, "img",
                    IO_RDWR|IO_EXST, 0);
    if (chndl < 0)
        goto done;

    clsbnd = get_bands(chndl, &nbandc);
    if (clsbnd == NULL)
        goto done;
    if (nbandc < MAXCLS)        /* Add new bands as required */
    {
        free((char *) clsbnd);
        if (add_bands(chndl, MAXCLS-nbandc, 0))
            goto done;
        clsbnd = get_bands(chndl, &nbandc);
        if (clsbnd == NULL)
            goto done;
    }
    chdr = get_header(chndl, 0);
    if (chdr == NULL)
        goto done;
/*
 *  Insure geometric compatibility
 */
    match = geom_match(ihndl, chndl, &roff, &coff, &isect);
    if (match < 0)
        goto done;

    if (!match || !isect)
    {
        put_string("Image and class map have different geometries\n");
        put_string("or do not intersect\n");
        set_error(E_LOGIC);
        goto done;
    }

    row1 = max(ihdr->row_start, chdr->row_start-roff);
    nrows = min(ihdr->row_start+ihdr->row_size,
                chdr->row_start+chdr->row_size-roff) - row1;
    col1 = max(ihdr->col_start, chdr->col_start-coff);
    ncols = min(ihdr->col_start+ihdr->col_size,
                chdr->col_start+chdr->col_size-coff) - col1;
```

Main routine for mixed pixel classifier

```
/*
 *  Get window to process
 */
   clear_screen();
   if (ask_window(-1, "Process entire area", &row1, &nrows, &col1,
                 &ncols))
       goto done;

   put_string("\n\n");
   if (ask_float("Enter maximum least squared error", &maxerr, 1, 0,
                V_GEMIN, &zero, (float *) NULL))
       goto done;
   maxerr *= maxerr;

   if (ask_float("\nEnter minimum allowable linear weight", &minwgt, 1,
                0, 0, (float *) NULL, (float *) NULL))
       goto done;
/*
 *  Allocate buffers
 */
   buff = (Pixel *) allocate((3*(MAXCLS + ncomm)) *
                        (ncols + 2) * sizeof(Pixel));
   if (buff == NULL)
       goto done;
/*
 *  Divide the area into individual buffers
 */
   for (n = 0; n < 3; n++)
   {
       for (m = 0; m < MAXCLS; m++)
       {
           clsbuf[n][m] = buff;
           buff += ncols + 2;
       }
   }
/*
 *  Allocate memory for image buffer pointers
 */
   imgbuf = (Pixel **) allocate(ncomm * 3 * sizeof(Pixel **));
   if (imgbuf == NULL)
       goto done;

   for (n = 0; n < (ncomm * 3); n++)
   {
        imgbuf[n] = buff;
        buff += ncols + 2;
    }
/*
 *  Start iterating and continue until no new pixels are classified
 */
    for (iter = 0, nchange = 1; nchange; iter++)
    {
        put_string("Starting pass %d\n", iter+1);
        stats.sum = 0.;
        stats.sumsq = 0.;
        stats.minalpha = 9999999.;
        stats.maxalpha = -9999999.;
        if (iterate(ihndl, chndl, cmbnd, ncomm, clsbuf, imgbuf, row1,
                    nrows, roff, col1, ncols, coff, mean, clsbnd,
                    &nchange, maxerr, minwgt, &stats, nclass))
           goto done;
        put_string("Pass %d, %d pixels classified\n", iter+1, nchange);
        if (nchange)
        {
        alpha_mean = stats.sum / (double) nchange;
        alpha_sd    =
                sqrt ((stats.sumsq - (nchange*(alpha_mean*alpha_mean))) / nchange);
        put_string("Alpha mean = %.2f, s.d. = %.2f, mn = %.2f, mx = %.2f\n",
                alpha_mean, alpha_sd, stats.minalpha, stats.maxalpha);
        }
    }

done:
    close_image(IO_ALL);
    exit(ioerror());
}
```

Function add_bands for mixed pixel classifier

```
#include <string.h>
#include "estypes.h"

int add_bands(hndl, ntoadd, fill)
/*
 * Add new bands to an image
 */
    int         hndl,        /* Image handle */
                ntoadd,      /* # of bands to add */
                fill;        /* Fill value for new bands */
{
    Pixel       *buff;
    HEADER      *hdr;
    FDB         *fdb;
    int         row, maxbnd, band, savacc, n;
    char        *period, name[129];

    if (ntoadd <= 0)
        return(0);
/*
 * Find max band #. Number new bands sequentially from this
 */
    hdr = get_header(hndl, 0);
    if (hdr == NULL)
        return(1);
/*
 * Get the fdb
 */
    fdb = get_fdb(hndl);
    if (fdb == NULL)
        return(1);

    maxbnd = hdr->band[0].band_num;
    for (band = 1; band < hdr->num_bands; band++)
    {
        if (hdr->band[band].band_num > maxbnd)
            maxbnd = hdr->band[band].band_num;
    }
/*
 * Reallocate per band area of header
 */
    hdr->band = (struct band_info *) reallocate(hdr->band,
                (hdr->num_bands + ntoadd) * sizeof(struct band_info));
    if (hdr->band == NULL)
```

```
        return(1);
/*
 * Fill in new band numbers and names
 */
    maxbnd++;
    file_name(fdb->header_name, name, 65);
    period = strrchr(name, '.');
    if (period != NULL)
        *period = '\0';                    /* Terminate at last period */
    (void) strcat(name, ".b");
    period = &name[strlen(name)];          /* period points to end */

    for (band = 0; band < ntoadd; band++)
    {
        n = hdr->num_bands + band;
        (void)sprintf(period, "%02d", n+1);
        (void)strcpy(fdb->header.band[n].band_name, name);
        hdr->band[n].band_num = maxbnd + band;
    }
/*
 * Rewrite ASCII header
 */
    hdr->num_bands += ntoadd;
    if (updt_header(hndl))
        return(1);
/*
 * Save old access mode and set access to WRITE with create
 */
    savacc = fdb->access;
    fdb->access = IO_WRITE;
/*
 * Allocate a buffer
 */
    buff = (Pixel *) allocate(hdr->col_size * sizeof(Pixel));
    if (buff == NULL)
        return(1);
/*
 * Fill with the fill value
 */
    for (n = 0; n < hdr->col_size; n++)
        buff[n] = fill;
/*
 * Create the bands
 */
```

```
    for (band = 0; band < ntoadd; band++)
    {
        if (set_bands(hndl, 1, &maxbnd))
            return(1);
        for (row = 0; row < hdr->row_size; row++)
        {
            if (write_image(hndl, maxbnd, row+hdr->row_start,
                            hdr->col_start, hdr->col_size, buff))
                return(1);
        }
        maxbnd++;
    }
/*
 *  Restore the access
 */
    fdb->access = savacc;
/*
 *  Free the buffer
 */
    free((char *) buff);
    return(0);
}
```

Function iterate for mixed pixel classifier

```
#include <memory.h>
#include "estypes.h"
#include "defs.h"

int iterate(ihndl, chndl, band, nband, clsbuf, imgbuf, row1, nrows,
            roff, col1, ncols, coff, mean, clsbnd, newpix, maxerr,
            minwgt, stats, totalclasses)
/*
 * Perform one iteration of the classification process
 */
   int       ihndl,            /* Image handle     */
             chndl,            /* Class map handle       */
            *band,                 /* Band numbers     */
             nband,            /* Number of bands        */
             row1,                 /* 1st row to process          */
             nrows,            /* # of rows to process */
             col1,             /* 1st column to process */
             ncols,            /* # of columns to process */
             roff,             /* Row offset for class map */
             coff,             /* Column offset for class map
                                */
            *clsbnd,           /* Class map band #'s       */
            *newpix;           /* # of pixels newly classified
                                */
   Pixel    *clsbuf[3][MAXCLS],    /* Buffer area                 */
            *imgbuf[];         /* Image buffer area        */
   float    *mean,                 /* Class means                 */
             maxerr,           /* Max ls error     */
             minwgt;           /* Min linear weight        */
      struct
      {
             double sum;
             double sumsq;
             double minalpha;
             double maxalpha;
      } *stats;
      int    totalclasses;  /* Total # pure classes */
{
   int       bnum,                 /* Current band #  */
             row,              /* Current row             */
             newcls;           /* # classified this line */
   Pixel    *temp;
   static
   float    *pixel = NULL,         /* Buffer for pixel values
```

```
                                         of center pixel */
            *clsval;                   /* Buffer for class values of
                                        * neighbors (nband rows by
                                        * MAXCLS cols) */
  char      *allocate();
/*
 *  On 1st pass, allocate space for pixel vector and clsval array
 */
  if (pixel == NULL)
  {
     pixel = (float *) allocate(nband * (MAXCLS + 1) *
                               sizeof(float));
     if (pixel == NULL)
        return(1);
     clsval = &pixel[nband];
  }
/*
 *  Prepare for I/O
 */
  if (set_bands(chndl, MAXCLS, clsbnd))
     return(1);
  if (set_bands(ihndl, nband, band))
     return(1);
/*
 *  Prime buffers by repeating 1st line
 */
  for (bnum = 0; bnum < MAXCLS; bnum++)
  {
     if (read_image(chndl, clsbnd[bnum], row1+roff, col1+coff, ncols,
                   &clsbuf[0][bnum][1]) < 0)
        return(1);
     clsbuf[0][bnum][0] = clsbuf[0][bnum][1];
     clsbuf[0][bnum][ncols+1] = clsbuf[0][bnum][ncols];
     (void) memcpy((char *) clsbuf[1][bnum],
                  (char *) clsbuf[0][bnum], ncols + 2);
  }
  for (bnum = 0; bnum < nband; bnum++)
  {
     if (read_image(ihndl, band[bnum], row1, col1, ncols,
                   &imgbuf[bnum][1]) < 0)
        return(1);
     imgbuf[bnum][0] = imgbuf[bnum][1];
     imgbuf[bnum][ncols+1] = imgbuf[bnum][ncols];
     (void) memcpy((char *) imgbuf[nband+bnum],
                    (char *) imgbuf[bnum], ncols + 2);
  }
/*
 * Cycle through the rows
 */
  for (row = *newpix = 0; row < nrows; row++)
  {
      count_down(row, 0, nrows);
/*
 * Read the image and class map bands unless this is the last line.
 * Else repeat last line.
 */
      if (row < (nrows - 1))
      {
         for (bnum = 0; bnum < MAXCLS; bnum++)
         {
            if (read_image(chndl, clsbnd[bnum], row1+roff+row+1,
                        col1+coff, ncols, &clsbuf[2][bnum][1])
                                < 0)
               return(1);
            clsbuf[2][bnum][0] = clsbuf[2][bnum][1];
            clsbuf[2][bnum][ncols+1] = clsbuf[2][bnum][ncols];
         }
         for (bnum = 0; bnum < nband; bnum++)
         {
            if (read_image(ihndl, band[bnum], row1+row+1, col1,
                        ncols, &imgbuf[2*nband+bnum][1]) < 0)
               return(1);
            imgbuf[2*nband+bnum][0] = imgbuf[2*nband+bnum][1];
            imgbuf[2*nband+bnum][ncols+1] = imgbuf[2*nband+bnum]
                                              [ncols];
         }
      }
      else
      {
         for (bnum = 0; bnum < MAXCLS; bnum++)
         {
            (void) memcpy((char *) clsbuf[2][bnum],
                        (char *) clsbuf[1][bnum], ncols + 2);
         }
         for (bnum = 0; bnum < nband; bnum++)
         {
            (void) memcpy((char *) imgbuf[2*nband+bnum],
                        (char *) imgbuf[nband+bnum], ncols + 2);
            }
        }
        if (clsline(nband, clsbuf, imgbuf, ncols, mean, pixel, clsval,
                &newcls, maxerr, minwgt, stats, totalclasses))
            return(1);
/*
 *  If pixels newly classified, rewrite classmap line
 */
        if (newcls)
        {
            *newpix += newcls;
            for (bnum = 0; bnum < MAXCLS; bnum++)
            {
                if (write_image(chndl, clsbnd[bnum], row1+roff+row,
                        col1+coff, ncols, &clsbuf[1][bnum][1])
                            < 0)
                    return(1);
            }
        }
/*
 *  Cycle line pointers
 */
        for (bnum = 0; bnum < MAXCLS; bnum++)
        {
            temp = clsbuf[0][bnum];
            clsbuf[0][bnum] = clsbuf[1][bnum];
            clsbuf[1][bnum] = clsbuf[2][bnum];
            clsbuf[2][bnum] = temp;
        }
        for (bnum = 0; bnum < nband; bnum++)
        {
            temp = imgbuf[bnum];
            imgbuf[bnum] = imgbuf[nband+bnum];
            imgbuf[nband+bnum] = imgbuf[2*nband+bnum];
            imgbuf[2*nband+bnum] = temp;
        }
    }

    return(0);
}
```

Function clsline for mixed pixel classifier

```
#include "estypes.h"
#include "defs.h"

#define errdump

int clsline(nband, clsbuf, imgbuf, ncols, mean, pixel, clsval,
        newpix, maxerr, minwgt, stats, totalclasses)
/*
 * Perform 1 line of the classification
 *
 * Modified to flag a pixel with > MAXCLS surrounding neighbors
 * as unclassifiable. Modifications also included to flag
 * failures of lsfit as unclassifiable (NULL_CLASS) and to sort
 * the resultant class assignments by their weights. This is
 * purely asthetic for simpified output purposes only as ALL
 * class assignment info (except for linear combination weights)
 * are maintained.
 */
   int        nband,              /* Number of bands        */
              ncols,              /* # of columns to process */
             *newpix;             /* # of pixels newly classified
                                   */
   Pixel     *clsbuf[3][MAXCLS],       /* Buffer area              */
             *imgbuf[];           /* Image buffer area      */
   float     *mean,                    /* Class means              */
             *pixel,                   /* Buffer for pixel values
                                    of center pixel */
             *clsval,             /* Buffer for class values of
                                   * neighbors (nband rows by
                                   * MAXCLS cols) */
              maxerr,             /* Max allowable ls error sq */
              minwgt;             /* Min allowable linear weight
                                   */
             struct
             {
                    double sum;
                    double sumsq;
                    double minalpha;
                    double maxalpha;
             } *stats ;
       int   totalclasses; /* total number of pure classes */
{
       double        sum;    /* Sum of alphas */
   int          newcls,            /* # of newly classified pixels */
```

```
            nclass,             /* # neighbor classes       */
            ntype[MAXCLS],   /* Neighbor type (+ =pure,0=mix)
                                */
            clsnum[MAXCLS], /* Neighbor class numbers */
            band,                       /* Current band #   */
            nghbr,              /* Neighbor index   */
            col,                /* Current column   */
            cnum, class, n, search, success;
  static
  int       rloc[] = {0, 1, 1, 2}, /* Row locations of neighbors */
            cloc[] = {1, 0, 2, 1}; /* Column locations        */
  float    *ptr, *ptr2,
            alpha[MAXCLS],   /* Linear weights    */
            error, diff;

  newcls = 0;                /* No pixels clasified yet */
/*
 * Cycle through the image columns
 */
  for (col = 0; col < ncols; col++)
  {
/*
 * If already classified go to next
 */
     if (clsbuf[1][0][col+1])
        continue;
/*
 * Cycle through the neighbors
 */
     for (nghbr = nclass = 0; (nghbr < MAXNBR)&&(nclass <= MAXCLS); nghbr++)
     {
/*
 * If neighbor not classified, goto next
 */
        if (!clsbuf[rloc[nghbr]]
                 [0]
                 [col+cloc[nghbr]])
           continue;
/*
 * If neighbor is a NULL CLASS pixel, ignore it.
 */
           if (clsbuf[rloc[nghbr]]
                    [0]
                    [col+cloc[nghbr]] == NULL_CLASS)
```

```
                continue;
/*
 * Neighbor is classified. Pure or mixed ?
 */
            if (clsbuf[rloc[nghbr]]
                    [1]
                    [col+cloc[nghbr]])   /* Mixed */
            {
                for (cnum = 0; cnum < MAXCLS; cnum++)
                {
                    class = clsbuf[rloc[nghbr]]
                                [cnum]
                                [col+cloc[nghbr]];
                    if (!class)
                        break;
/*
 * Check to see if we already have this class
 */
                    for (n = 0; n < nclass; n++)
                    {
                        if (clsnum[n] == class)
                            break;
                    }
/*
 * Did we find it ?
 */
                    if (n == nclass)
                    {
                        if (nclass == MAXCLS)
/*
 * Flag this pixel as no more visits necessary!
 */
                        {
                                clsbuf[1][0][col+1] = NULL_CLASS;
                                nclass++;  /* Flag too many classes */
                                newcls++;
                            break;     /* Too many neighbor classes */
                        }

                        clsnum[nclass] = class;
                        ptr = &clsval[nclass * nband];
                        ptr2 = &mean[(class-1)*nband];
                        for (band = 0; band < nband; band++)
                            ptr[band] = ptr2[band];
                    ntype[nclass] = 0;
                    nclass++;
                 }
              }
           }
           else                    /* Pure */
           {
/*
 * Check to see if we already have this class
 */
              class = clsbuf[rloc[nghbr]]
                          [0]
                          [col+cloc[nghbr]];
              for (n = 0; n < nclass; n++)
              {
                 if (clsnum[n] == class)
                 {
/*
 * We have it but was it mixed ?
 */
                    if (!ntype[n])
                    {
                       ptr = &clsval[n * nband];
                       for (band = 0; band < nband; band++)
                       {
                          ptr[band] =
                           (float) imgbuf[rloc[nghbr]*nband+band]
                                  [col+cloc[nghbr]];
                       }
                       ntype[n] = 1;
                    }
/*
 * We have found another occurrence of a pure class. Add the pixel
 * values and increment count. We'll normalize later
 */
                    else
                    {
                       ptr = &clsval[n * nband];
                       for (band = 0; band < nband; band++)
                       {
                          ptr[band] +=
                           (float) imgbuf[rloc[nghbr]*nband+band]
                                  [col+cloc[nghbr]];
                       }
```

```
                        ntype[n]++;
                    }
                    break;
                }
            }
/*
 *  Did we find it ?
 */
            if (n == nclass)
            {
                if (nclass == MAXCLS)
                    {
                            clsbuf [1][0][col+1] = NULL_CLASS;
                            nclass++; /* Flag too many classes */
                            newcls++;
                    continue;       /* Too many neighbor classes */
                    }

                clsnum[nclass] = class;
                ptr = &clsval[nclass * nband];
                for (band = 0; band < nband; band++)
                {
                    ptr[band] =
                        (float) imgbuf[rloc[nghbr]*nband+band]
                                    [col+cloc[nghbr]];
                }
                ntype[nclass] = 1;
                nclass++;
            }
        }
    }
/*
 *  Have we found between 1 and MAXCLS neighbor classes ?
 */
    if ((nclass > 0) && (nclass <= MAXCLS))
    {
/*
 *  Extract this pixel's vector
 */
        for (band = 0; band < nband; band++)
                pixel[band] = (float) imgbuf[nband+band][col+1];
/*
 *  Normalize any multiple occurrence pure classes
 */
```

```
        for (class = 0; class < nclass; class++)
        {
                if (ntype[class] > 1)
                {
                ptr = &clsval[class * nband];
                for (band = 0; band < nband; band++)
                        ptr[band] /= ntype[class];
                }
        }

/*
 * If 2 <= NCLASSES <= MAXCLS try a strict linear combination.  If
 * it fails and there are <= MAXCLS-1 classes, see if there is
 * another "pure" class mixed in by iterratively adding a
 * class not occuring in this neighborhood already.
 */
            if (nclass == 1)
                    search = 1;   /* Flag search necessary */
            else
            {                                  /* Attempt classification with neighbors only */
            if (lsfit(nband, nclass, clsval, pixel, alpha))
                    {
                            printf("lsfit returned non-zero.");
                            errdump (nband, nclass, clsval, pixel);
                            continue;  /* with next pixel */
                    }

                    search = error_check (nband, nclass, clsval, pixel, alpha,
                            minwgt, maxerr);
            }
/*
 * If we didn't have success with a simple linear combination of the
 * neighborhood OR if we only had 1 surrounding class, substitute
 * in classes not included in the original neighborhood to see if we
 * can get an acceptable fit using another (possibly sub-pixel level)
 * known pure class using the class means.
 */
            if (!search)
                    success = 1;  /* Previous classification successful */
            else
            {
                    success = 0;
                    if (nclass < MAXCLS)
                    {
```

```
                    nclass++;   /* We'll have one more class now */
                    for (class = 1 ; class <= totalclasses ; class++)
                    {
                            for (cnum = 0 ; cnum < nclass-1 ; cnum++)
                                    if (clsnum [cnum] == class)
                                            break;
/*
 *  If we found a non-included class, try it with the neighborhood
 */
                            if (cnum == (nclass-1))  /* Ah-ha, found one */
                            {
                                    clsnum [nclass-1] = class;
                                    ptr = &clsval [(nclass-1)*nband];
                                    ptr2= &mean [(class-1)*nband];
                                    for (band = 0 ; band < nband ; band++)
                                            ptr [band] = ptr2 [band];
                                    ntype [nclass-1] = 0;   /* (Mixed pixel */

                                    if (lsfit (nband, nclass, clsval, pixel, alpha))
                                    {
                                            printf("lsfit returned non-zero.\n");
                                            errdump(nband, nclass, clsval, pixel);
                                            break;
                                    }

                                    if (!error_check (nband, nclass, clsval, pixel,
                                                alpha, minwgt, maxerr))
                                    {
                                            success = 1;
                                            break;
                                    }
                            }
                    }
            }
    }

    if (success)
    {
/*
 *  Passed the test. Store class numbers in classmap bands but
 *  only if the linear weight for that class was > 0
 */
            sortcls (nclass,clsnum,alpha);
            sum = 0.;
```

Function clsline for mixed pixel classifier

```
                for (class = n = 0; class < nclass; class++)
                        if (alpha[class] > 0.0)
                        {
                                sum += alpha [class];
                        clsbuf[1][n++][col+1] = clsnum[class];
                        }
                if (n)
                                newcls++;
                        stats->sum += (double) sum;
                        stats->sumsq += ((double) sum * (double) sum);
                        if (stats->minalpha > sum)
                                stats->minalpha = sum;
                        if (stats->maxalpha < sum)
                                stats->maxalpha = sum;
                }
                }
        }
        *newpix = newcls;
        return(0);
}
```

Function smatinv for mixed pixel classifier

```
#define   abs(x)   (((x) < 0) ? (x) : -(x))

int smatinv(a, n, list)
/*
 *  Invert a single precision matrix
 */
   float        *a;            /* before: matrix to invert,
                                  after: inverted matrix */
   int          n,             /* nxn matrix */
                *list;         /* Scratch vector (n elements) */
{
   float        detr, abval, b, div;
   int          irno, icon,
                i, ir, j, k;

   detr = 1.0;

   for (i = 0; i < n; i++)
       list[i] = i;

   for (ir = 0; ir < n; ir++)
   {
       irno = ir;
       if (ir < (n - 1))
       {
           div = abs(a[ir*n+ir]);
           j = ir + 1;

           for (i = j; i < n; i++)
           {
               abval = abs(a[ir*n+i]);
               if ((div - abval) < 0.0)
               {
                   div = abval;
                   irno = i;
               }
           }

           if (irno != ir)
           {
/*
 *   interchange row ir and irno
 */
               for (j = 0; j < n; j++)
```

```
                {
                    b = a[j*n+irno];
                    a[j*n+irno] = a[j*n+ir];
                    a[j*n+ir] = b;
                }

                detr = -detr;
                icon = list[irno];
                list[irno] = list[ir];
                list[ir] = icon;
            }
        }

        div = a[ir*n+ir];
        if (div == 0.0)
            return(1);

        detr *= div;
        a[ir*n+ir] = 1.0;
        for (j = 0; j < n; j++)
            a[j*n+ir] /= div;

        for (j = 0; j < n; j++)
        {
            if (j != ir)
            {
                b = a[ir*n+j];
                a[ir*n+j] = 0.0;
                for (k = 0; k < n; k++)
                    a[k*n+j] -= b * a[k*n+ir];
            }
        }
    }
/*
 *   interchange columns
 */
    for (k = 0; k < n; k++)
    {
        for (i = k; i < n; i++)
            if (list[i] == k)
                break;

        icon = list[k];
        list[k] = list[i];
        list[i] = icon;
        for (j = 0; j < n; j++)
        {
            b = a[k*n+j];
            a[k*n+j] = a[i*n+j];
            a[i*n+j] = b;
        }
    }
    return(0);
}
```

Function sortcls for mixed pixel classifier

```
#include "estypes.h"
#include "defs.h"

int
sortcls (nclass, clsnum, alpha)
        int     *clsnum;
        int   nclass;
        float *alpha;
{

        int       i,j,temp2;
        float temp;
/*
 * Begin code
 */
        for (i = 0 ; i < (nclass-1) ; i++)
        for (j = (i+1) ; j < nclass ; j++)
        {
                if (alpha [i] < alpha [j])
                {
                        temp      = alpha [i];
                        alpha [i] = alpha [j];
                        alpha [j] = temp;
                        temp2     = clsnum [i];
                        clsnum[i] = clsnum [j];
                        clsnum[j] = temp2;
                }
        }
}
```

Function lsfit for mixed pixel classifier

```
#include <stdio.h>
#include "eserror.h"
#include "defs.h"

int lsfit(nband, nclass, neighbor, pixel, alpha)
/*
 * Compute least squares solution of the vector of weights
 * for each spectral class surrounding the pixel.
 */
   int        nband,      /* # of spectral bands           */
              nclass;     /* # of classes              */
   float      *neighbor,  /* nband row by nclass column    *
                           * matrix of spectral means of   *
                           * neighboring classes           */
              *pixel,     /* Spectral band values for      *
                           * center pixel              */
              *alpha;     /* Vector of weights for each    *
                           * class (returned)          */
{
   static
   int        *list = NULL;
   static
   float      *ntn, *mat;
   int        class, band,
              class1;
   char       *allocate();
   float      *ptr1, *ptr2, *ptr3;
/*
 * 1st time through, allocate scratch arrays
 */
   if (list == NULL)
   {
      ntn = (float *) allocate(MAXCLS*(MAXCLS+nband)*sizeof(float) +
                             MAXCLS*sizeof(int));
      if (ntn == NULL)
         return(1);
      mat = &ntn[MAXCLS*MAXCLS];
      list = (int *) &mat[MAXCLS*nband];
   }
/*
 * Compute product of neighbor matrix and its transpose
 */
   ptr1 = neighbor;
   for (class = 0; class < nclass; class++)
   {
      ptr2 = neighbor;
      ptr3 = &ntn[class];

      for (class1 = 0; class1 < nclass; class1++)
      {
         ptr3[0] = 0.0;
         for (band = 0; band < nband; band++)
             ptr3[0] += ptr1[band] * ptr2[band];

         ptr3 += nclass;
         ptr2 += nband;
      }
      ptr1 += nband;
   }
/*
 * Invert this
 */
   if (smatinv(ntn, nclass, list))
   {
      set_error(E_LOGIC);
      return(1);
   }
/*
 * Multiply the transpose of neighbor by the inverted matrix
 */
   ptr1 = ntn;
   for (class = 0; class < nclass; class++)
   {
      ptr2 = neighbor;
      ptr3 = &mat[class];

      for (band = 0; band < nband; band++)
      {
         ptr3[0] = 0.0;
         for (class1 = 0; class1 < nclass; class1++)
             ptr3[0] += ptr1[class1*nclass] * ptr2[class1*nband];

         ptr3 += nclass;
         ptr2++;
      }
      ptr1++;
   }
/*
```

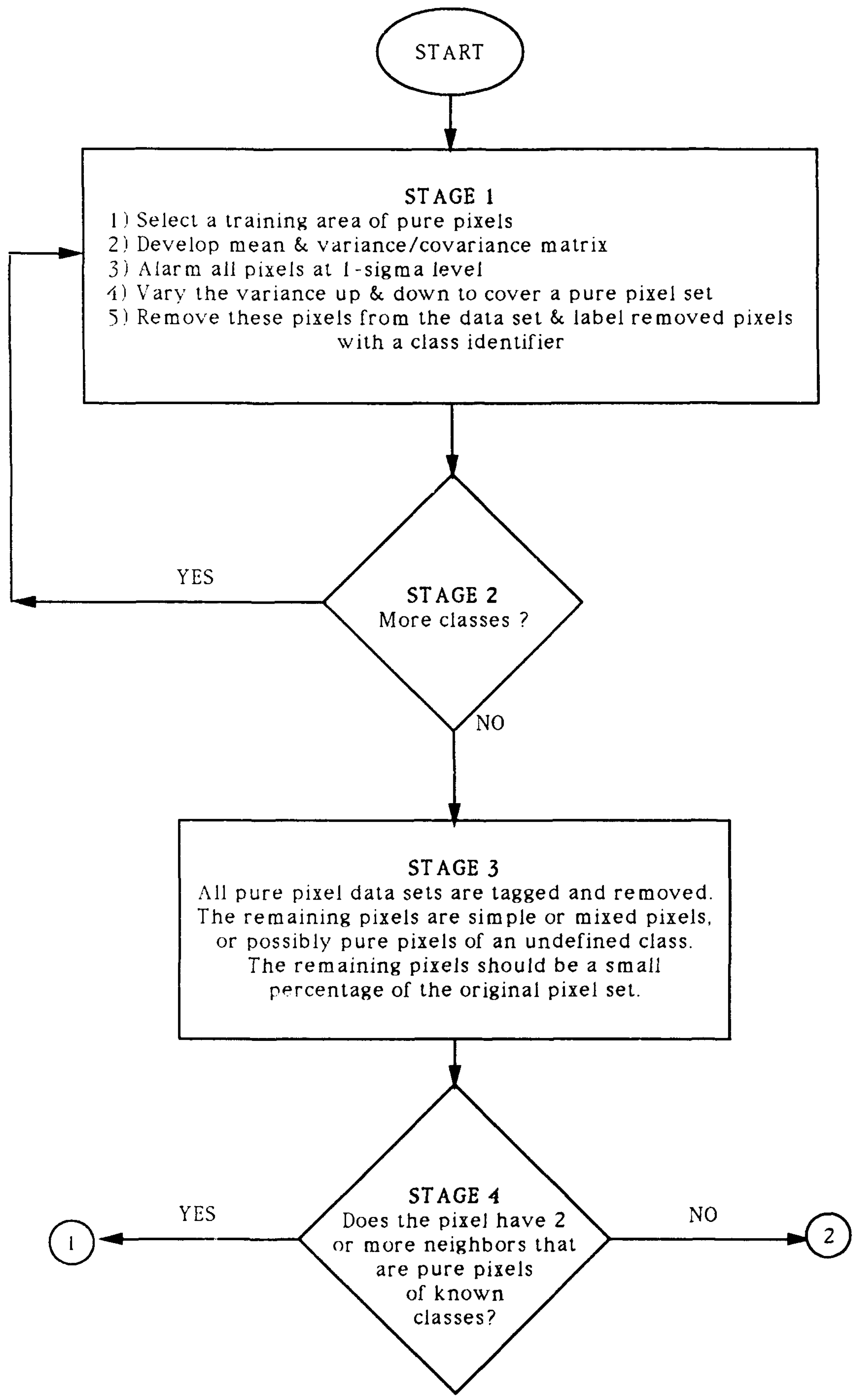
Function lsfit for mixed pixel classifier

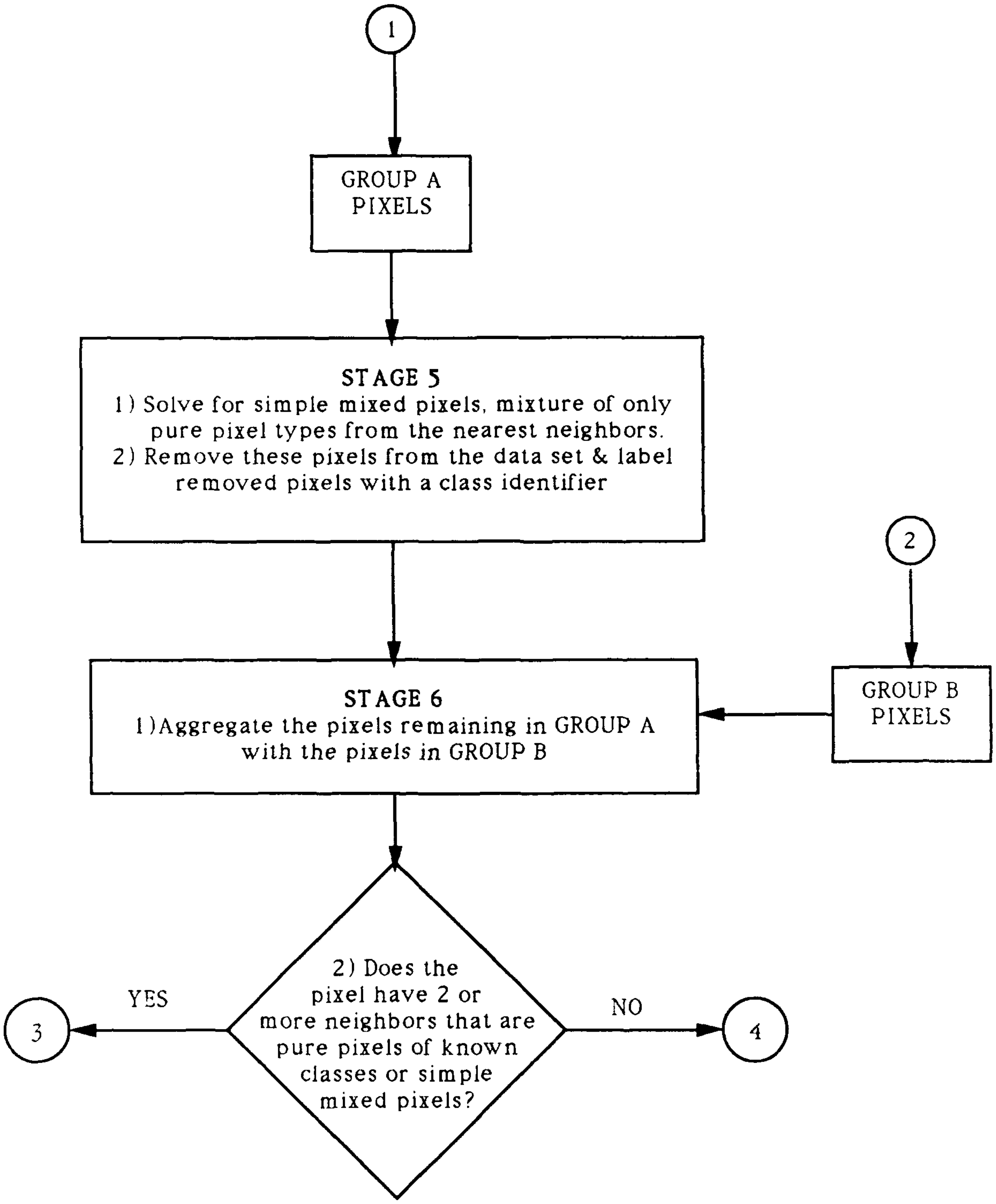
```
 *  Generate alpha weights by multiplying pixel vector by mat
 */
   for (class = 0; class < nclass; class++)
   {
       alpha[class] = 0.0;

       for (band = 0; band < nband; band++)
           alpha[class] += mat[class+band*nclass] * pixel[band];
   }
    return(0);
}
```

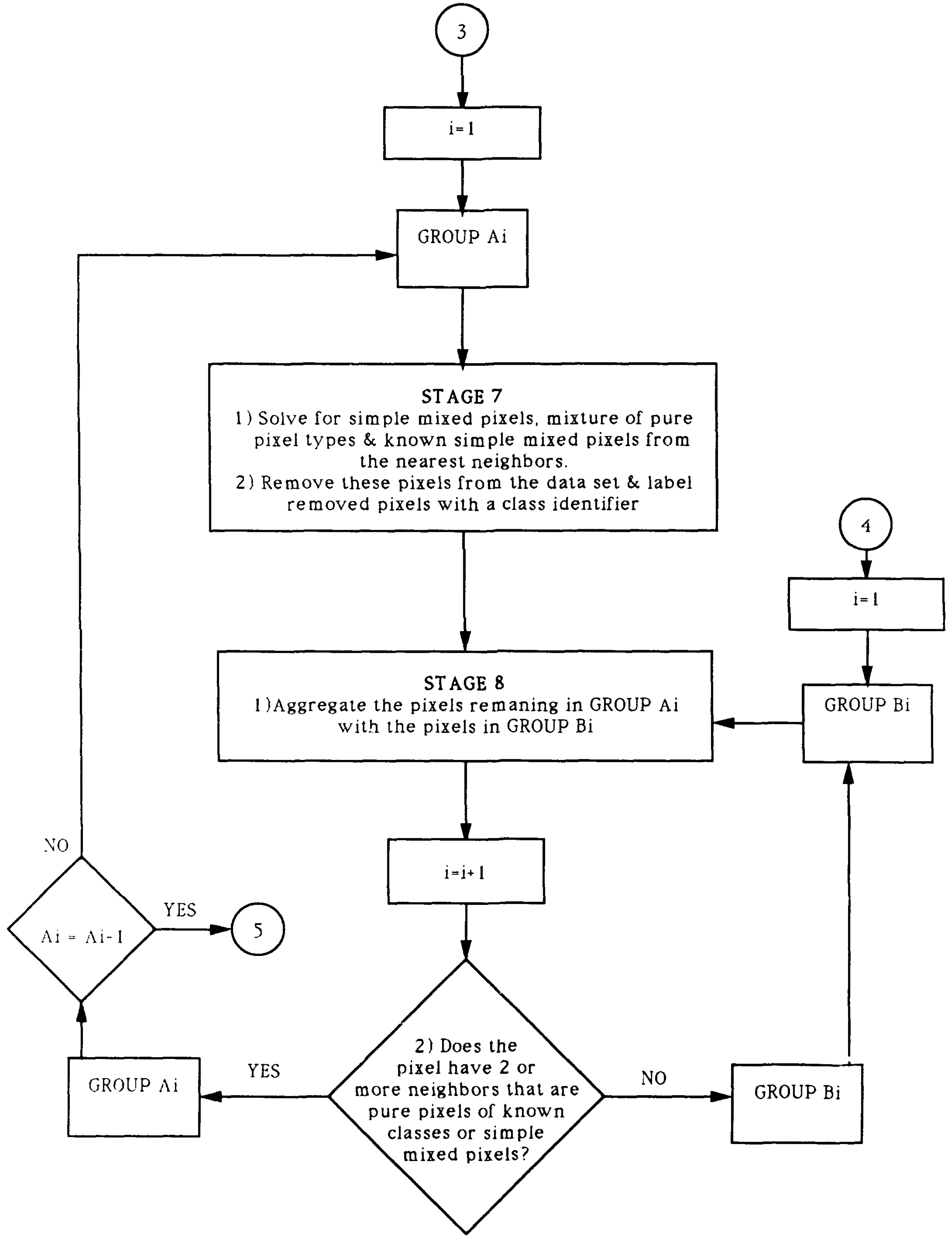
Function error_check for mixed pixel classifier

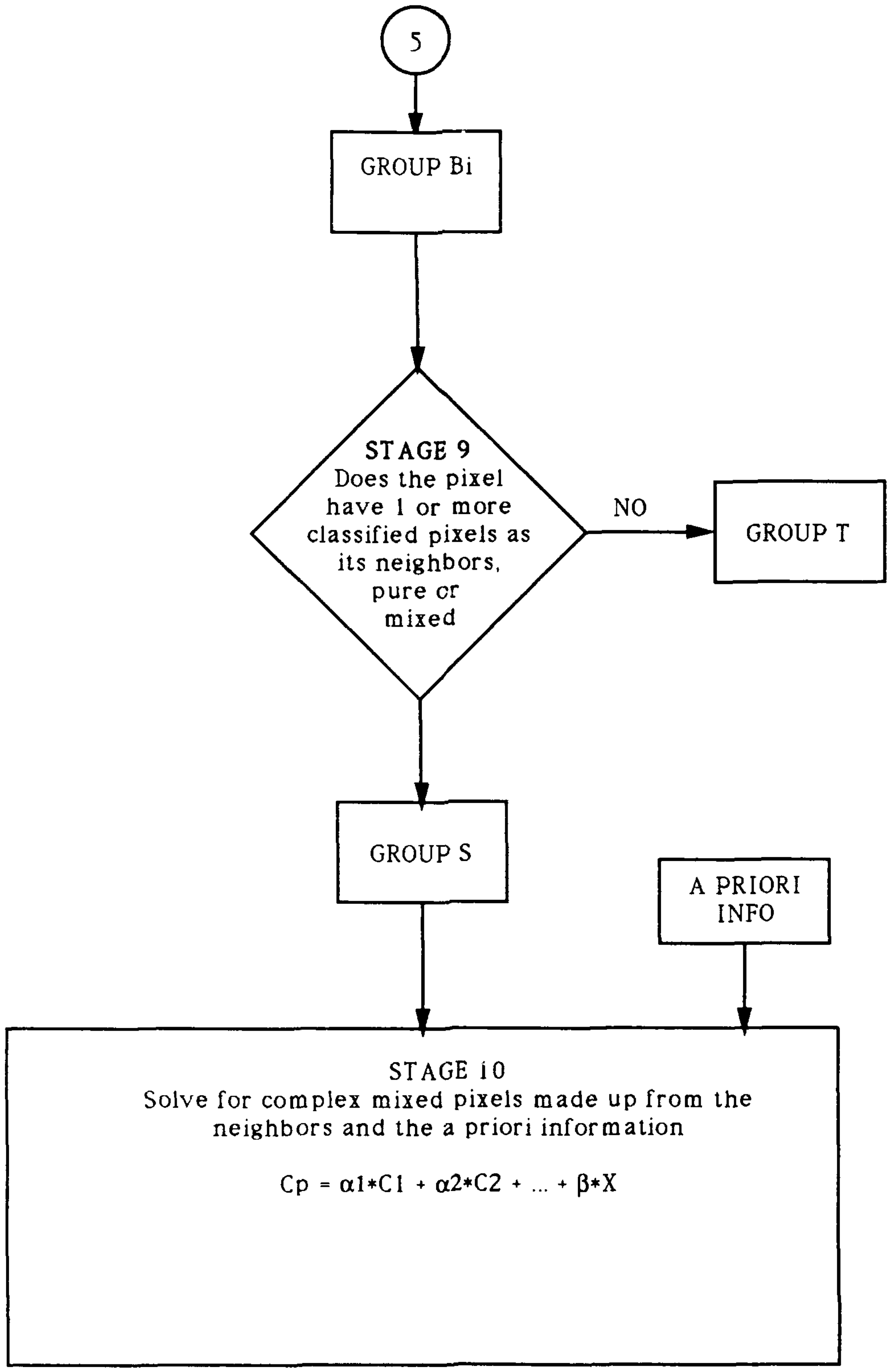
```
#include "estypes.h"
#include "defs.h"

int
error_check (nband, nclass, clsval, pixel, alpha, minwgt, maxerr)
        float *pixel;
        float *alpha;
        float *clsval;
        int   nclass;
        float minwgt;
        float maxerr;
{

        int       class,
                  band;
        float error,
                  diff;
/*
 * Begin code
 */
        for (class = 0 ; class < nclass ; class++)
                if (alpha [class] < minwgt)
                        alpha [class] = 0.;

        error = 0.;
        for (band = 0 ; band < nband ; band ++)
        {
                diff = -pixel [band];
                for (class = 0 ; class < nclass ; class++)
                        diff += alpha [class] * clsval [class*nband+band];

                error += diff * diff;
        }

        if (error > maxerr)
                return (1);

        return (0);
}
```

GENERAL FLOW LOGIC FOR MIXED PIXEL CLASSIFICATION


START
STAGE 1
1) Select a training area of pure pixels
2) Develop mean & variance/covariance matrix
3) Alarm all pixels at 1-sigma level
4) Vary the variance up & down to cover a pure pixel set
5) Remove these pixels from the data set & label removed pixels with a class identifier
YES
STAGE 2
More classes ?
NO
STAGE 3
All pure pixel data sets are tagged and removed. The remaining pixels are simple or mixed pixels, or possibly pure pixels of an undefined class. The remaining pixels should be a small percentage of the original pixel set.
YES
STAGE 4
Does the pixel have 2 or more neighbors that are pure pixels of known classes?
NO
1
2


1
GROUP A PIXELS
STAGE 5
1) Solve for simple mixed pixels, mixture of only pure pixel types from the nearest neighbors.
2) Remove these pixels from the data set & label removed pixels with a class identifier
2
GROUP B PIXELS
STAGE 6
1)Aggregate the pixels remaining in GROUP A with the pixels in GROUP B
2) Does the pixel have 2 or more neighbors that are pure pixels of known classes or simple mixed pixels?
YES
3
NO
4


3
i=1
GROUP Ai
STAGE 7
1) Solve for simple mixed pixels, mixture of pure pixel types & known simple mixed pixels from the nearest neighbors.
2) Remove these pixels from the data set & label removed pixels with a class identifier
4
i=1
GROUP Bi
STAGE 8
1)Aggregate the pixels remaning in GROUP Ai with the pixels in GROUP Bi
NO
i=i+1
YES
Ai = Ai-1
5
2) Does the pixel have 2 or more neighbors that are pure pixels of known classes or simple mixed pixels?
YES
GROUP Ai
NO
GROUP Bi


5
GROUP Bi
STAGE 9
Does the pixel have 1 or more classified pixels as its neighbors, pure or mixed
NO
GROUP T
GROUP S
A PRIORI INFO
STAGE 10
Solve for complex mixed pixels made up from the neighbors and the a priori information
Cp = α1*C1 + α2*C2 + ... + β*X

APPENDIX 2: CONTRACT DELIVERABLES NOT INCLUDED AS PART OF THIS REPORT

The following is a list of the separately delivered photographic products:

Item #	Quantity	Description
1-5	2 each	Color print of 2k x 2k source mosaic in band combination 247 (BGR) at a scale of 1:100,000 for five scene dates (May 1985, August 1985, October 1985, May 1987, and March 1989)
6-10	2 each	Color print of 2k x 2k source mosaic in band combination 274 (BGR) at a scale of 1:100,000 for five scene dates (May 1985, August 1985, October 1985, May 1987 and March 1989)
11-15	2 each	Color print of 2k x 2k source mosaic in band combination 354 (BGR) at a scale of 1:100,000 for five scene dates (May 1985, August 1985, October 1985, May 1987 and March 1989)
16-20	2 each	Color print of 2k x 2k source mosaic in band combination 234 (BGR) at a scale of 1:100,000 for five scene dates (May 1985, August 1985, October 1985, May 1987 and March 1989)
21-25	2 each	Color print of 2k x 2k source mosaic in band combination 123 (BGR) at a scale of 1:100,000 for five scene dates (May 1985, August 1985, October 1985, May 1987 and March 1989)
26-30	2 each	Color print of 2k x 2k source mosaic in a Tasseled Cap transform at a scale of 1:100,000 for five scene dates (May 1985, August 1985, October 1985, May 1987 and March 1989)
31-32	2 each	Color print of 2k x 2k source mosaic of classification at a scale of 1:100,000 for scene dates May 1985 and May 1987
33	2 total	Color print of subscene change detection between May 1985 and May 1987 using a maximum likelihood classification at 3.0 standard deviations

34-36	2 each	Color print of subscene change detection between May 1985 and May 1987 using a mixed pixel classification at a least squares tolerance of 15 pixels, 10 pixels and 5 pixels
37-39	2 each	Set of 2 color prints of subscene mixed pixel classification at a least squares tolerance of 15 pixels, 10 pixels and 5 pixels for a scene date of May 1985
40-42	2 each	Set of 2 color prints of subscene mixed pixel classification at a least squares tolerance of 15 pixels, 10 pixels and 5 pixels for a scene date of May 1987
43	2 total	Black and White image of principal component urban vector image with atmospheric, vegetation and intensity components as a composite of all scene dates
44-79	2 each	Color transparency of Items #1-36
80-81	2 each	Color transparency of subscene mixed pixel classification at least squares tolerance of 15, 10 and 5 pixels for scene date of May 1985 and May 1987
82-83	2 each	Color transparency of subscene mixed pixel classification 4 level classmap with maximum likelihood classification of 3.0 standard deviations for scene dates May 1985 and May 1987

The following is the list of separately delivered magnetic media products:

85	1	5-1/4 inch floppy disk of the final report in WordPerfect Version 5.1 format.
86	1	Magnetic tape of the mixed pixel source code developed for this report in unlabeled ASCII 9 track 1/2 inch 6250 bpi format
87	1	Magnetic tape of the subscene classifications done for this report in unlabeled sequential 9 track 1/2 inch 6250 bpi format

APPENDIX 3. Projection operators and band combinations

1. Projection along a given axis

We assume we are working with 6-band TM data. Then any pixel in a scene has the grey levels in all 6 bands described by a 6-vector, **g**.

Given any other (fixed) unit 6-vector, **P**, a new band can be created, corresponding to projection along the axis of **P**, in which each pixel has the new grey level, **P . g.**

This new band can be displayed, for any choice of **P**, and is uniquely defined when **P** is specified. **P** is chosen, usually interactively, to maximize (or minimize) some chosen image property.

2. Projection perpendicular to a given axis

Given the unit vector **P**, we can consider it as an axis in a rotated space, relative to the original 6-space of the TM data. There will then be a 5-space orthogonal to **P**. However, even when **P** is given, the 5-space orthogonal to it may be defined in many different ways.

One common method uses Gram-Schmidt orthogonalization. With this technique, if the original band axes are $\mathbf{e}_1 \ldots \mathbf{e}_6$, then 6 new axes, each orthogonal to **P**, can be defined by forming

$$\mathbf{e}^1_i = \mathbf{P} - (\mathbf{e}_i / \mathbf{e}_i.\mathbf{P})$$

These new axes are not orthogonal, and they are also not linearly independent.

However, a set of 5 independent and orthogonal bands can be created from them in the 5-space orthogonal to **P** by forming

$$\mathbf{E}_1 = \mathbf{e}_1^1 / |\mathbf{e}_1^1|$$

where $|\mathbf{e}_1^1|$ = length of $\mathbf{e}_1^1$ = $\sqrt{e_1^1 \cdot e_1^1}$

$$\mathbf{E}_2 = (\mathbf{E}_1 - (\mathbf{e}_2^1/\mathbf{E}_1.\mathbf{e}_2^1))/|\mathbf{E}_1 - (\mathbf{e}_2^1/\mathbf{E}_1.\mathbf{e}_2)|$$

$$\mathbf{E}_3 = (\mathbf{E}_1 - (\mathbf{e}_2^1/\mathbf{E}_1.\mathbf{e}_2^1) - (\mathbf{e}_3^1/\mathbf{E}_1.\mathbf{e}_3^1))/|\mathbf{E}_1 - (\mathbf{e}_2^1/\mathbf{E}_1.\mathbf{e}_2^1) - (\mathbf{e}_3^1/\mathbf{E}_1.\mathbf{e}_3^1)|$$

and so on up to $\mathbf{E}_5$

3. Further projection operators

We now have a defined direction, **P**, and a set of 5 bands orthogonal to **P** and to each other.

We can now therefore choose a second vector, **Q**, in this 5-space, and project along it as before, by forming $\mathbf{Q} \cdot \mathbf{g}^1$, where now the grey levels $\mathbf{g}^1$ are those in the five-space defined by $\mathbf{E}_1 \ldots \mathbf{E}_5$, orthogonal to **P**.

Having selected **Q** (by definition orthogonal to **P**), a new space orthogonal to **P** and **Q** can be defined by a Gram-Schmidt orthogonalization in the 5-space $\mathbf{E}_1 \ldots \mathbf{E}_5$. As before, we form a set of vectors.

$$\mathbf{f}_i = \mathbf{Q} - \mathbf{E}_i/\mathbf{E}_i.\mathbf{Q}$$

and use them to build 4 independent and orthogonal bands, $\mathbf{F}_i$, by $\mathbf{F}_1 = \mathbf{f}_1/|\mathbf{f}_1|$, etc.

The whole procedure may be repeated for projection vectors **R**, **S**...until a total of 6 orthogonal bands, each a linear combination of the original 6 bands, has been found.